

## Contrast Enhancement

### 1. Image Negative

Image negative inverts the pixel intensity values. Dark pixels become bright and bright pixels become dark. This is a linear point processing operation applied to each pixel.

$$s = (L - 1) - r$$

for 8 bit image  $L = 255$

#### Meaning:

$r$  = input pixel intensity

$s$  = output pixel intensity

$L$  = number of gray levels (256)

#### Code:

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg', 0)
negative = 255 - img

plt.subplot(1,2,1);
plt.imshow(img, cmap='gray');
plt.title("Original");
plt.axis('off')

plt.subplot(1,2,2);
plt.imshow(negative, cmap='gray');
plt.title("Negative");
plt.axis('off')
plt.show()
```

### 2. Log Transformation

Log transformation expands low intensity pixels and compresses high intensity pixels. It is useful when important details are present in dark regions.

$$s = c * \log(1 + r)$$

#### Scaling constant:

$$c = \frac{255}{\log(1 + r_{max})}$$

#### Code:

```

import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg', 0)
c = 30
log_img = c * np.log1p(img)
log_img = np.clip(log_img, 0, 200).astype('uint8')

plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(log_img, cmap='gray')
plt.title("Log")
plt.axis('off')
plt.show()

```

### 3. Power Law \Gamma Transformation

Gamma correction adjusts brightness non-linearly. If gamma < 1 the image becomes brighter, and if gamma > 1 the image becomes darker.

$$s = c * r^{\text{gamma}}$$

*Normalized form*

$$s = 255 * \left(\frac{r}{255}\right)^{\text{gamma}}$$

**Code:**

```

import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg', 0)
gamma = 5
gamma_img = (255*(img/255)**gamma).astype('uint8')

plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(gamma_img, cmap='gray')
plt.title("Gamma")
plt.axis('off')
plt.show()

```

### 4. Contrast Stretching

Contrast stretching linearly remaps pixel intensities from a narrow range to the full range [0,255] to improve image contrast.

$$s = \left( \frac{r - r_{min}}{r_{max} - r_{min}} \right) * 255$$

$r_{min}$  = minimum pixel value

$r_{max}$  = maximum pixel value

### Code:

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg', 0)

stretch = cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX)

plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(stretch, cmap='gray')
plt.title("Auto Stretch (OpenCV)")
plt.axis('off')

plt.show()
```

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg', 0)

r_min, r_max = img.min(), img.max()

stretch = (img - r_min) * (255/(r_max - r_min))
stretch = np.clip(stretch, 0, 255).astype('uint8')

plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(1,2,2)
plt.imshow(stretch, cmap='gray')
plt.title("Auto Stretch")
plt.axis('off')

plt.show()
```

## 5. Histogram

Histogram represents the frequency distribution of gray levels in an image. It shows how many pixels exist for each intensity value.

$$p(r_k) = \frac{n_k}{M * N}$$

$n_k$  = number of pixels with intensity  $r_k$

$M * N$  = total number of pixels

$p(r_k)$  = probability of intensity level

**Code:**

```
import cv2
import matplotlib.pyplot as plt

img = cv2.imread('image.jpg', 0)

plt.subplot(1,2,1)
plt.imshow(img, cmap='gray')
plt.title("Image")
plt.axis('off')

plt.subplot(1,2,2)
plt.hist(img.ravel(), bins=256, range=[0,256])
plt.title("Histogram")
plt.show()
```

## 6. Histogram Equalization

Histogram equalization enhances contrast by redistributing pixel intensities using the cumulative distribution function (CDF). It is a non-linear global contrast enhancement method.

**Mathematical Steps:**

Step 1:  $p(r_k) = \frac{n_k}{M * N}$

Step 2:  $CDF(r_k)$  = sum of probabilities up to level  $k$

Step 3:  $s_k = (L - 1) * CDF(r_k)$

Intensity ( $r_k$ )	Frequency ( $n_k$ )	Probability $P(r_k) = \frac{n_k}{N}$	CDF $S(r_k)$	Equalized Value $s_k = (L - 1) \times CDF$
0	1	0.0625	0.0625	0
1	2	0.1250	0.1875	1
2	2	0.1250	0.3125	2
3	3	0.1875	0.5000	4
4	2	0.1250	0.6250	4
5	2	0.1250	0.7500	5
6	2	0.1250	0.8750	6
7	2	0.1250	1.0000	7

## Code:

```
import cv2
import matplotlib.pyplot as plt

# Read image
img = cv2.imread('image.jpg', 0)

# Equalization
eq = cv2.equalizeHist(img)

plt.figure(figsize=(10,6))

plt.subplot(2,2,1)
plt.imshow(img, cmap='gray')
plt.title("Original Image")
plt.axis('off')

plt.subplot(2,2,2)
plt.hist(img.ravel(), bins=256, range=[0,256])
plt.title("Original Histogram")

plt.subplot(2,2,3)
plt.imshow(eq, cmap='gray')
plt.title("Equalized Image")
plt.axis('off')

plt.subplot(2,2,4)
plt.hist(eq.ravel(), bins=256, range=[0,256])
plt.title("Equalized Histogram")

plt.tight_layout()
plt.show()
```

## Manual Code

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Read image in grayscale
img = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

if img is None:
    print("Image not found. Check file path.")
    exit()

# Step 2: Calculate histogram
hist = np.zeros(256)

for row in img:
    for pixel in row:
        hist[pixel] += 1
```

```

# Step 3: Calculate CDF (Cumulative Distribution Function)
cdf = hist.cumsum()

# Normalize CDF
cdf_normalized = (cdf - cdf.min()) * 255 / (cdf.max() - cdf.min())
cdf_normalized = cdf_normalized.astype('uint8')

# Step 4: Map original pixels using CDF
equalized_img = cdf_normalized[img]

# Step 5: Display results
plt.figure(figsize=(12,8))

plt.subplot(2,2,1)
plt.title("Original Image")
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(2,2,2)
plt.title("Equalized Image")
plt.imshow(equalized_img, cmap='gray')
plt.axis('off')

plt.subplot(2,2,3)
plt.title("Original Histogram")
plt.hist(img.flatten(), bins=256, range=[0,256])

plt.subplot(2,2,4)
plt.title("Equalized Histogram")
plt.hist(equalized_img.flatten(), bins=256, range=[0,256])

plt.tight_layout()
plt.show()

```

## All Comparison

```

import cv2
import numpy as np
import matplotlib.pyplot as plt

# ----- Read Image -----
img = cv2.imread('image.jpg', 0)

if img is None:
    print("Error: Image not found")
    exit()

# 1. Negative Transformation
negative = 255 - img

# 2. Log Transformation
c = 30 # you can change
log_img = c * np.log1p(img)
log_img = np.clip(log_img, 0, 255).astype('uint8')

```

```

# 3. Gamma Transformation
gamma = 0.5 # you can change
gamma_img = (255 * (img / 255) ** gamma).astype('uint8')

# 4. Contrast Stretching

r_min, r_max = img.min(), img.max()
stretch = (img - r_min) * (255 / (r_max - r_min))
stretch = np.clip(stretch, 0, 255).astype('uint8')

# 5. Histogram Equalization

equalized = cv2.equalizeHist(img)

#PLOT
plt.figure(figsize=(15,10))

# Row 1
plt.subplot(3,3,1)
plt.imshow(img, cmap='gray')
plt.title("Original")
plt.axis('off')

plt.subplot(3,3,2)
plt.imshow(negative, cmap='gray')
plt.title("Negative")
plt.axis('off')

plt.subplot(3,3,3)
plt.imshow(log_img, cmap='gray')
plt.title("Log Transform")
plt.axis('off')

# Row 2
plt.subplot(3,3,4)
plt.imshow(gamma_img, cmap='gray')
plt.title("Gamma")
plt.axis('off')

plt.subplot(3,3,5)
plt.imshow(stretch, cmap='gray')
plt.title("Contrast Stretch")
plt.axis('off')

plt.subplot(3,3,6)
plt.imshow(equalized, cmap='gray')
plt.title("Histogram Equalized")
plt.axis('off')

# Row 3 - Histograms
plt.subplot(3,3,7)
plt.hist(img.ravel(), bins=256, range=[0,256])
plt.title("Original Histogram")

plt.subplot(3,3,8)
plt.hist(equalized.ravel(), bins=256, range=[0,256])

```

```
plt.title("Equalized Histogram")

plt.subplot(3,3,9)
plt.axis('off')

plt.tight_layout()
plt.show()
```



# MATLAB

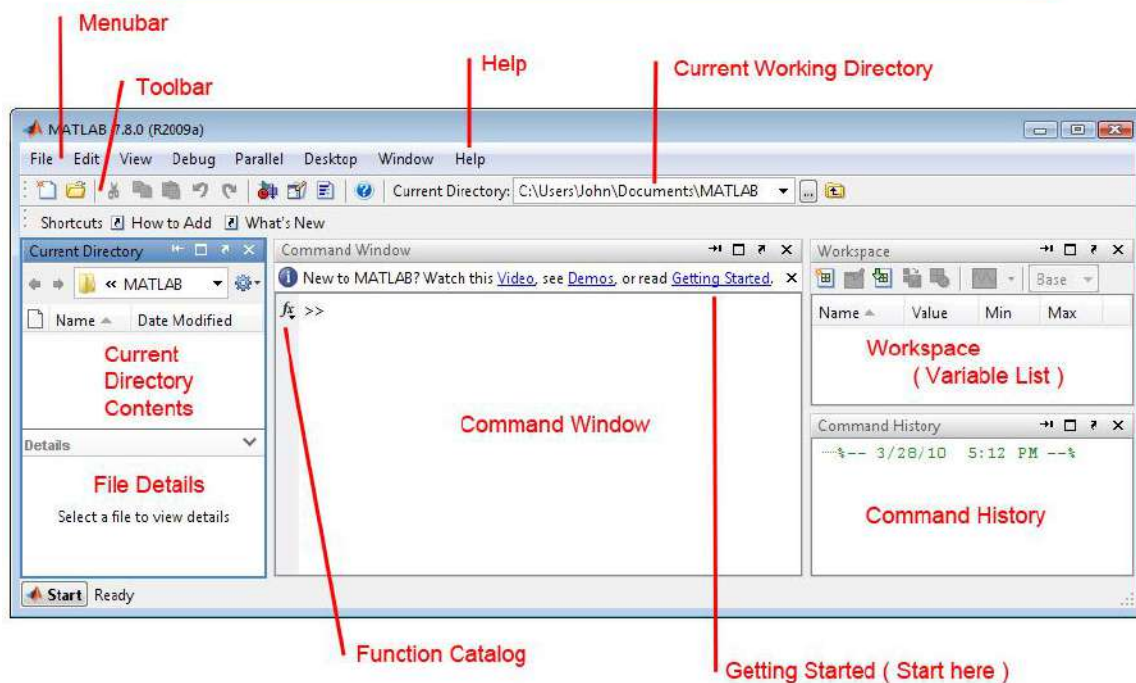
## What is MATLAB?

**MATLAB (Matrix Laboratory)** is a programming and numerical computing environment used for:

- Mathematical calculations
- Data analysis
- Signal & image processing
- Control systems
- Machine learning & simulations

It is **matrix-based**, meaning most operations work naturally on arrays and matrices.

## The MATLAB Work Environment



## Main Parts

- **Command Window** → type and run commands
- **Workspace** → shows current variables
- **Editor** → write and save scripts/functions

- **Figure Window** → displays plots/graphs
- **Current Folder** → file navigation

#### ◇ **Workspace & Environment Commands**

Command	Purpose
clc	Clear Command Window
clear	Remove all variables
clear x	Remove specific variable
close all	Close all figure windows
who	List variables
whos	Detailed variable info
pwd	Show current folder path
cd foldername	Change directory
dir	List files in folder

`disp('Hello MATLAB')`

`x = 10;`

`fprintf('Value of x = %d\n', x)`

#### ◇ **Mathematical Commands**

Command	Meaning
sqrt(x)	Square root
abs(x)	Absolute value
exp(x)	$e^x$
log(x)	Natural log
log10(x)	Base-10 log
sin(x), cos(x), tan(x)	Trigonometry
round(x)	Round value
floor(x)	Round down
ceil(x)	Round up

*sqrt(25)*

*sin(pi/2)*

*log10(100)*

*A = [1 2 3; 4 5 6]*

## Special matrices

Command	Result
<code>zeros(3,3)</code>	3×3 zero matrix
<code>ones(2,4)</code>	Matrix of ones
<code>eye(3)</code>	Identity matrix
<code>rand(3,3)</code>	Random matrix
<code>linspace(a,b,n)</code>	n points between a and b

---

*clc*

*clear*

*close all*

*x = 0:0.1:2 \* pi;*

*y = sin(x);*

*plot(x,y)*

*title('Basic MATLAB Practice')*

*grid on*

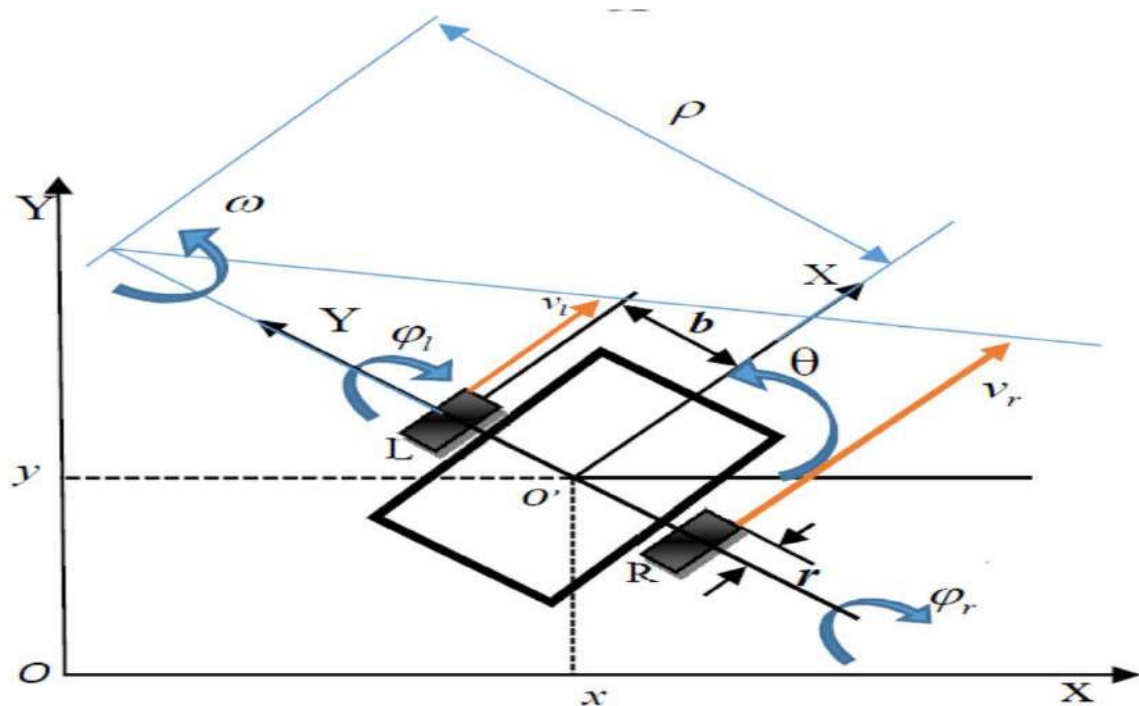
## 🚲 Unicycle System — Detailed Explanation

### 1 What is a Unicycle Model?

The unicycle model is a nonlinear non-holonomic kinematic model of a mobile robot whose motion is controlled by linear and angular velocities. Also can be define as “The **unicycle model** is a mathematical representation of a mobile robot that moves like a single-wheel vehicle”.

It is used because many real robots behave similarly, such as:

- Differential drive robots
- Two-wheel robots
- Mobile service robots
- Some autonomous vehicle models (simplified)



### 2 Key Motion Characteristics

A unicycle robot:

- ✓ Moves forward/backward
- ✓ Rotates about its center
- ✗ Cannot move sideways directly

Because of this sideways restriction, it is called a **non-holonomic system**.

### 3 Coordinate System

We describe the robot in a **2D plane**.

**Robot pose (state):**

$$q = (x, y, \theta)$$

Where:

- $x \rightarrow$  X position in global frame
  - $y \rightarrow$  Y position in global frame
  - $\theta \rightarrow$  orientation (heading angle)
- 

### ◇ Orientation Meaning

- $\theta = 0 \rightarrow$  facing positive X
  - $\theta = \pi/2 \rightarrow$  facing positive Y
  - $\theta = \pi \rightarrow$  facing negative X
- 

### 4 Control Inputs

The unicycle has **two control inputs**:

**Linear velocity**

$$v(\text{m/s})$$

- Controls forward/backward motion
  - Applied along robot heading
- 

**Angular velocity**

$$\omega(\text{rad/s})$$

- Controls turning
- Changes orientation

## 5 Kinematic Model (Core Equations)

The unicycle motion is governed by:

$$\dot{x} = v \cos \theta$$

$$\dot{y} = v \sin \theta$$

$$\dot{\theta} = \omega$$

---

### Physical Interpretation

#### Equation 1: X motion

$$\dot{x} = v \cos \theta$$

Meaning:

- Robot moves in heading direction
  - Only X component is projected using cosine
- 

#### Equation 2: Y motion

$$\dot{y} = v \sin \theta$$

Meaning:

- Y motion depends on orientation
- 

#### Equation 3: Orientation change

$$\dot{\theta} = \omega$$

Meaning:

- Angular velocity directly rotates robot
- 

## 6 Non-Holonomic Constraint


The unicycle cannot move sideways.

Mathematically:

$$\dot{y} \cos \theta - \dot{x} \sin \theta = 0$$

This is called the **non-holonomic constraint** of the unicycle.

### What It Means

 It says:

The robot cannot move sideways.

In other words:

- Motion **perpendicular to the wheel** is zero
- The robot can only move **along its heading direction**

### Physical Intuition

Imagine a real bicycle or single wheel:

- It can roll forward/backward
- It can turn
- But it **cannot slide sideways**

This equation mathematically enforces that rule.

### What This Means Physically

Velocity perpendicular to wheel direction = **zero**

So, robot must:

✓ rotate

✓ then move forward

✗ cannot slide sideways

## Motion Behavior Cases

### Case 1: Straight Line Motion

If:

$$\omega = 0$$

Then:

- heading constant
- robot moves straight

Trajectory:

$$\begin{aligned}x(t) &= x_0 + vt \cos \theta \\y(t) &= y_0 + vt \sin \theta\end{aligned}$$

---

### Case 2: Pure Rotation

If:

$$v = 0, \omega \neq 0$$

Then:

- robot spins in place
  - position unchanged
- 

### Case 3: Circular Motion

If:

$$v \neq 0, \omega \neq 0$$

Robot moves in a circle.

Radius of circle:

$$R = \frac{v}{\omega}$$


### Why Circular?

Because:

- forward motion + turning  
→ produces arc trajectory

### Control Objective

In most problems we want:

 Move robot from

$$(x, y, \theta) \rightarrow (x_d, y_d)$$

Goal:



- minimize position error
  - minimize angle error
- 

## 9 Error Definitions (Used in Controllers)

### Position error

$$e_d = \sqrt{(x_d - x)^2 + (y_d - y)^2}$$

### Desired heading

$$\theta_d = \tan^{-1} \left( \frac{y_d - y}{x_d - x} \right)$$

### Angle error

$$e_\theta = \theta_d - \theta$$

## 10 Typical Control Law (Very Important)

Most unicycle controllers use:

### Linear velocity

$$v = k_1 \cdot e_d$$

### Angular velocity

$$\omega = k_2 \cdot e_\theta$$

Where:

- $k_1, k_2$  are gains
- can be P / PI / PD / PID

## 11 Discrete-Time Implementation

In simulation (MATLAB/Python):

$$\begin{aligned}x_{k+1} &= x_k + v \cos \theta \cdot dt \\y_{k+1} &= y_k + v \sin \theta \cdot dt \\\theta_{k+1} &= \theta_k + \omega \cdot dt\end{aligned}$$

## 1 2 Applications

Unicycle model is widely used in:

- Mobile robot navigation
  - Path tracking
  - Formation control
  - SLAM research
  - Autonomous parking
  - PID control teaching
  - Reinforcement learning robotics
- 

## 1 3 Common Mistakes

- ✗ Treating it as holonomic
- ✗ Forgetting angle wrapping
- ✗ Using large gains  $\rightarrow$  oscillation
- ✗ Ignoring nonlinear nature
- ✗ Not normalizing angle to  $[-\pi, \pi]$

## PID Controller

**What is PID?**

PID (Proportional–Integral–Derivative) is a feedback control method used to minimize the error between a desired value (setpoint) and the measured output.

$$e(t) = r(t) - y(t)$$

*Where:*

- $r(t)$  = desired value (setpoint)
- $y(t)$  = actual output
- $e(t)$  = error

Control signal:

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

Where:

- $K_p \rightarrow$  proportional gain
- $K_i \rightarrow$  integral gain
- $K_d \rightarrow$  derivative gain

## 1. Proportional (P) Controller

Proportional controller reacts to the **present error only**.

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

**Working**

- Large error  $\rightarrow$  strong control
- Small error  $\rightarrow$  weak control
- Zero error  $\rightarrow$  no action

### Example

Setpoint = 25°C

Measured = 20°C

Error = 5

If  $K_p = 2$ :

$$u = 2 \times 5 = 10$$

Heater power = 10 units.

### Advantages

- ✓ Fast response
- ✓ Simple design

### Disadvantages

- ✗ Cannot remove steady-state error
- ✗ High  $K_p$  causes overshoot
- ✗ Possible oscillations

### Effect of Increasing $K_p$

Kp	Effect
small	slow
medium	good
large	oscillatory

## 2. Integral (I) Controller

### ◆ Concept

Integral controller reacts to **accumulated past error**.

$$u(t) = K_i \int_0^t e(\tau) d\tau$$

### ◆ Purpose

**remove steady-state error**

### ◆ Working Intuition

If error persists for long time → keep increasing control.

### ◆ Example

Setpoint = 50°C

Measured stuck at = 48°C

Error = 2 (constant)

### Integral accumulation

Time	Integral	Output
1 s	2	$2K_i$

5 s	10	10 <i>K<sub>i</sub></i>
10 s	20	20 <i>K<sub>i</sub></i>

### Advantages

- ✓ Eliminates steady-state error
- ✓ Improves accuracy

### ◆ Disadvantages

- ✗ Slow response
- ✗ Causes overshoot
- ✗ May cause oscillations
- ✗ Integral windup

### ⚠ Integral Windup

When actuator saturates, integral keeps growing → large overshoot later.

### Prevention

- integral clamping
- anti-windup
- conditional integration

### Effect of Increasing $K_i$

K <sub>i</sub>	Effect
small	slow correction
medium	good
large	unstable

## 3. Derivative (D) Controller

Derivative reacts to **rate of change of error**.

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

### ◆ Purpose

- predicts future error
- adds damping
- reduces overshoot

#### ♦ Intuition

Acts like a **brake**.

If error changing fast → apply opposite action.

#### ♦ Example

Previous error = 10

Current error = 4

dt = 1 s

$$\frac{de}{dt} = -6$$

If  $K_d = 2$ :

$$u_D = -12$$

Controller slows the system.

#### ♦ Advantages

- ✓ Reduces overshoot
- ✓ Improves stability
- ✓ Faster settling

#### ♦ Disadvantages

- ✗ Sensitive to noise
- ✗ Cannot remove steady-state error
- ✗ Rarely used alone

#### ⚠ Noise Issue

Derivative amplifies measurement noise.

#### Practical fix

Use filtered derivative:

$$\frac{K_d s}{\tau s + 1}$$

◆ **Effect of Increasing  $K_d$**

$K_d$	Effect
small	little damping
medium	good
large	noisy / sluggish

#### 4. Combined PID Controller

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

◆ **Role of Each Term**

Term	Role
P	present error
I	past error
D	future trend

#### Overall Effect

- ✓ Fast rise time
- ✓ Zero steady-state error
- ✓ Reduced overshoot
- ✓ Good stability

### PID Implementation for Unicycle System

This is very important for robotics and drone/UGV work.

◆ **Unicycle Kinematic Model**

State variables:

- position:  $x, y$
- orientation:  $\theta$

Model:

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

Where:

- $v$  = linear velocity (control input)
- $\omega$  = angular velocity (control input)

### ◆ Control Objective

Drive robot from current pose:

$$(x, y, \theta)$$

to desired pose:

$$(x_d, y_d, \theta_d)$$

### ◆ Step 1: Compute Errors

**Position error**

$$e_{pos} = \sqrt{(x_d - x)^2 + (y_d - y)^2}$$

**Heading error**

$$e_{\theta} = \theta_d - \theta$$

(usually normalized to  $-\pi$  to  $\pi$ )

### ◆ Step 2: Control Law

**Linear velocity (usually PI)**

$$v = K_{p,v} e_{pos} + K_{i,v} \int e_{pos} dt$$

**Angular velocity (PID)**

$$\omega = K_{p,\omega} e_{\theta} + K_{i,\omega} \int e_{\theta} dt + K_{d,\omega} \frac{de_{\theta}}{dt}$$



---

### ◆ Step 3: Discrete Implementation (MATLAB Style)

Assume sampling time = dt.

**% --- position error ---**

$e_{pos} = \sqrt{(x_d - x)^2 + (y_d - y)^2};$

**% --- heading error ---**

$e_{\theta} = \text{wrapToPi}(\theta_d - \theta);$

**% --- integral update ---**

$e_{pos\_int} = e_{pos\_int} + e_{pos} \cdot dt;$

$e_{\theta\_int} = e_{\theta\_int} + e_{\theta} \cdot dt;$

**% --- derivative ---**

$e_{\theta\_der} = (e_{\theta} - e_{\theta\_prev}) / dt;$

**% --- control laws ---**

$v = K_p \cdot e_{pos} + K_i \cdot e_{pos\_int};$

$\omega = K_p \cdot e_{\theta} + K_i \cdot e_{\theta\_int} + K_d \cdot e_{\theta\_der};$

**% --- store previous ---**

$e_{\theta\_prev} = e_{\theta};$

---

### ◆ Step 4: System Update

$x = x + v \cdot \cos(\theta) \cdot dt;$

$y = y + v \cdot \sin(\theta) \cdot dt;$

$\theta = \theta + \omega \cdot dt;$

---

### Without I term

✗ robot stops near target but not exactly

### With I term

✓ removes steady-state position error

### Without D term

✗ oscillatory turning

### With D term

✓ smooth orientation convergence

---

### Typical Gain Starting Point

(you must tune experimentally)

$K_{p_v} = 1.0$

$K_{i_v} = 0.1$

$K_{p_w} = 3.0$

$K_{i_w} = 0.05$

$K_{d_w} = 0.5$

### 🎯 Final Conclusion

PID control is powerful because:

- P gives fast reaction
- I remove steady-state error
- D reduces overshoot and improves stability

For mobile robots like the unicycle model, properly tuned PID enables **smooth, accurate, and stable navigation** to the target pose.

---


```
clc; clear; close all;
%% ===== COMMON SETTINGS =====
dt = 0.01;
```

```

T = 20;
t = 0:dt:T;

xd = 5;
yd = 5;

%% wrap helper inline usage
wrap = @(ang) mod(ang + pi, 2*pi) - pi;

%% =====
%%  P CONTROLLER
x=0; y=0; theta=0;
Kp_v=0.8; Kp_w=2;

x_traj=[]; y_traj=[]; err_P=[];

for k=1:length(t)

    ex=xd-x; ey=yd-y;
    rho=sqrt(ex^2+ey^2);
    theta_d=atan2(ey,ex);
    e_theta=wrap(theta_d-theta);

    v=Kp_v*rho;
    w=Kp_w*e_theta;

    x=x+v*cos(theta)*dt;
    y=y+v*sin(theta)*dt;
    theta=theta+w*dt;

    x_traj(k)=x;
    y_traj(k)=y;
    err_P(k)=rho;
end

figure;
subplot(2,1,1)
plot(xd,yd,'ro','MarkerSize',10,'LineWidth',2); hold on;
plot(x_traj,y_traj,'b','LineWidth',2);

```

```
title('P Controller Trajectory'); legend('Target','Robot');  
grid on; axis equal;
```

```
subplot(2,1,2)  
plot(t,err_P,'LineWidth',2);  
title('P Controller Error'); grid on;
```

```
%% =====
```

```
%%  I CONTROLLER
```

```
x=0; y=0; theta=0;  
Ki_v=0.05; Ki_w=0.1;  
int_v=0; int_w=0;
```

```
x_traj=[]; y_traj=[]; err_I=[];
```

```
for k=1:length(t)
```

```
    ex=xd-x; ey=yd-y;  
    rho=sqrt(ex^2+ey^2);  
    theta_d=atan2(ey,ex);  
    e_theta=wrap(theta_d-theta);
```

```
    int_v=int_v+rho*dt;  
    int_w=int_w+e_theta*dt;
```

```
    int_v=max(min(int_v,10),-10);  
    int_w=max(min(int_w,10),-10);
```

```
    v=Ki_v*int_v;  
    w=Ki_w*int_w;
```

```
    x=x+v*cos(theta)*dt;  
    y=y+v*sin(theta)*dt;  
    theta=theta+w*dt;
```

```
    x_traj(k)=x;  
    y_traj(k)=y;  
    err_I(k)=rho;
```

```
end
```

```
figure;
subplot(2,1,1)
plot(xd,yd,'ro','MarkerSize',10,'LineWidth',2); hold on;
plot(x_traj,y_traj,'b','LineWidth',2);
title('I Controller Trajectory'); legend('Target','Robot');
grid on; axis equal;
```

```
subplot(2,1,2)
plot(t,err_I,'LineWidth',2);
title('I Controller Error'); grid on;
```

```
%% =====
```

```
%%  D CONTROLLER
```

```
x=0; y=0; theta=0;
Kd_v=0.2; Kd_w=0.4;
prev_rho=0; prev_eth=0;
```

```
x_traj=[]; y_traj=[]; err_D=[];
```

```
for k=1:length(t)
```

```
    ex=xd-x; ey=yd-y;
    rho=sqrt(ex^2+ey^2);
    theta_d=atan2(ey,ex);
    e_theta=wrap(theta_d-theta);
```

```
    der_v=(rho-prev_rho)/dt;
    der_w=(e_theta-prev_eth)/dt;
```

```
    v=Kd_v*der_v;
    w=Kd_w*der_w;
```

```
    prev_rho=rho;
    prev_eth=e_theta;
```

```
    x=x+v*cos(theta)*dt;
    y=y+v*sin(theta)*dt;
    theta=theta+w*dt;
```

```

        x_traj(k)=x;
        y_traj(k)=y;
        err_D(k)=rho;
end

figure;
subplot(2,1,1)
plot(xd,yd,'ro','MarkerSize',10,'LineWidth',2); hold on;
plot(x_traj,y_traj,'b','LineWidth',2);
title('D Controller Trajectory'); legend('Target','Robot');
grid on; axis equal;

subplot(2,1,2)
plot(t,err_D,'LineWidth',2);
title('D Controller Error'); grid on;

%% =====
%% 🌀 PI CONTROLLER
x=0; y=0; theta=0;
Kp_v=0.8; Ki_v=0.05;
Kp_w=2;   Ki_w=0.1;
int_v=0; int_w=0;

x_traj=[]; y_traj=[]; err_PI=[];

for k=1:length(t)

    ex=xd-x; ey=yd-y;
    rho=sqrt(ex^2+ey^2);
    theta_d=atan2(ey,ex);
    e_theta=wrap(theta_d-theta);

    int_v=int_v+rho*dt;
    int_w=int_w+e_theta*dt;

    int_v=max(min(int_v,10),-10);
    int_w=max(min(int_w,10),-10);

```

```

v=Kp_v*rho + Ki_v*int_v;
w=Kp_w*e_theta + Ki_w*int_w;

x=x+v*cos(theta)*dt;
y=y+v*sin(theta)*dt;
theta=theta+w*dt;

x_traj(k)=x;
y_traj(k)=y;
err_PI(k)=rho;
end

figure;
subplot(2,1,1)
plot(xd,yd,'ro','MarkerSize',10,'LineWidth',2); hold on;
plot(x_traj,y_traj,'b','LineWidth',2);
title('PI Controller Trajectory'); legend('Target','Robot');
grid on; axis equal;

subplot(2,1,2)
plot(t,err_PI,'LineWidth',2);
title('PI Controller Error'); grid on;

%% =====
%% 🌀 PD CONTROLLER
x=0; y=0; theta=0;
Kp_v=0.8; Kd_v=0.2;
Kp_w=2; Kd_w=0.4;
prev_rho=0; prev_eth=0;

x_traj=[]; y_traj=[]; err_PD=[];

for k=1:length(t)

    ex=xd-x; ey=yd-y;
    rho=sqrt(ex^2+ey^2);
    theta_d=atan2(ey,ex);
    e_theta=wrap(theta_d-theta);

    der_v=(rho-prev_rho)/dt;

```

```

    der_w=(e_theta-prev_eth)/dt;

    v=Kp_v*rho + Kd_v*der_v;
    w=Kp_w*e_theta + Kd_w*der_w;

    prev_rho=rho;
    prev_eth=e_theta;

    x=x+v*cos(theta)*dt;
    y=y+v*sin(theta)*dt;
    theta=theta+w*dt;

    x_traj(k)=x;
    y_traj(k)=y;
    err_PD(k)=rho;
end

figure;
subplot(2,1,1)
plot(xd,yd,'ro','MarkerSize',10,'LineWidth',2); hold on;
plot(x_traj,y_traj,'b','LineWidth',2);
title('PD Controller Trajectory'); legend('Target','Robot');
grid on; axis equal;

subplot(2,1,2)
plot(t,err_PD,'LineWidth',2);
title('PD Controller Error'); grid on;

%% =====
%% 🌀🌀🌀 PID CONTROLLER
x=0; y=0; theta=0;
Kp_v=0.8; Ki_v=0.05; Kd_v=0.2;
Kp_w=2; Ki_w=0.1; Kd_w=0.4;
int_v=0; int_w=0;
prev_rho=0; prev_eth=0;

x_traj=[]; y_traj=[]; err_PID=[];

for k=1:length(t)

```



```

ex=xd-x; ey=yd-y;
rho=sqrt(ex^2+ey^2);
theta_d=atan2(ey,ex);
e_theta=wrap(theta_d-theta);

int_v=int_v+rho*dt;
int_w=int_w+e_theta*dt;

int_v=max(min(int_v,10),-10);
int_w=max(min(int_w,10),-10);

der_v=(rho-prev_rho)/dt;
der_w=(e_theta-prev_eth)/dt;

v=Kp_v*rho + Ki_v*int_v + Kd_v*der_v;
w=Kp_w*e_theta + Ki_w*int_w + Kd_w*der_w;

prev_rho=rho;
prev_eth=e_theta;

x=x+v*cos(theta)*dt;
y=y+v*sin(theta)*dt;
theta=theta+w*dt;

x_traj(k)=x;
y_traj(k)=y;
err_PID(k)=rho;
end

figure;
subplot(2,1,1)
plot(xd,yd,'ro','MarkerSize',10,'LineWidth',2); hold on;
plot(x_traj,y_traj,'b','LineWidth',2);
title('PID Controller Trajectory'); legend('Target','Robot');
grid on; axis equal;

subplot(2,1,2)
plot(t,err_PID,'LineWidth',2);
title('PID Controller Error'); grid on;

```

```

%% =====
%% 📊 FINAL OVERALL COMPARISON
figure;
plot(t,err_P,'LineWidth',2); hold on;
plot(t,err_I,'LineWidth',2);
plot(t,err_D,'LineWidth',2);
plot(t,err_PI,'LineWidth',2);
plot(t,err_PD,'LineWidth',2);
plot(t,err_PID,'LineWidth',3);

legend('P','I','D','PI','PD','PID');
title('Overall Controller Comparison');
xlabel('Time (s)');
ylabel('Distance Error');
grid on;

```

## Animation

```

clc; clear; close all;

%% ===== SETTINGS =====
dt = 0.05;
T = 20;
t = 0:dt:T;

xd = 5;
yd = 5;

% PID gains (stable for demo)
Kp_v=0.8; Ki_v=0.05; Kd_v=0.2;
Kp_w=2;   Ki_w=0.1;   Kd_w=0.4;

% initial state
x=0; y=0; theta=0;
int_v=0; int_w=0;
prev_rho=0; prev_eth=0;

wrap = @(ang) mod(ang + pi, 2*pi) - pi;

```

```

%% ===== FIGURE SETUP =====
figure;
axis equal;
grid on;
hold on;
xlim([-1 6]);
ylim([-1 6]);

title('Animated Unicycle PID Control');
xlabel('X'); ylabel('Y');

% target
plot(xd,yd,'rp','MarkerSize',14,'LineWidth',2);

% robot graphics
robot_body = plot(x,y,'bo','MarkerSize',10,'MarkerFaceColor','b');
robot_dir = quiver(x,y,cos(theta),sin(theta),0.5,'k','LineWidth',2);
traj_plot = plot(x,y,'b');

err_text = text(0.1,5.5,'Error: 0','FontSize',12,'Color','r');

%% ===== ANIMATION LOOP =====
for k=1:length(t)

    % ----- error -----
    ex=xd-x;
    ey=yd-y;
    rho=sqrt(ex^2+ey^2);
    theta_d=atan2(ey,ex);
    e_theta=wrap(theta_d-theta);

    % ----- PID -----
    int_v=int_v+rho*dt;
    int_w=int_w+e_theta*dt;

    int_v=max(min(int_v,10),-10);
    int_w=max(min(int_w,10),-10);

    der_v=(rho-prev_rho)/dt;
    der_w=(e_theta-prev_eth)/dt;

```

```

v=Kp_v*rho + Ki_v*int_v + Kd_v*der_v;
w=Kp_w*e_theta + Ki_w*int_w + Kd_w*der_w;

prev_rho=rho;
prev_eth=e_theta;

% ----- robot update -----
x=x+v*cos(theta)*dt;
y=y+v*sin(theta)*dt;
theta=theta+w*dt;

% ----- update graphics -----
set(robot_body, 'XData',x, 'YData',y);
set(robot_dir, 'XData',x, 'YData',y, ...
    'UData',cos(theta), 'VData',sin(theta));

% trajectory
xt = get(traj_plot, 'XData');
yt = get(traj_plot, 'YData');
set(traj_plot, 'XData',[xt x], 'YData',[yt y]);

% error text
set(err_text, 'String',sprintf('Error: %.3f',rho));

drawnow;

% stop early if reached
if rho < 0.05
    disp('☑ Target Reached');
    break;
end
end

```