DEVELOPMENT OF REAL-TIME DASHBOARD VISUALIZATION OF KEY PROCESS PARAMETERS

**Week 1: Learning and Research Phase**

During the first week, I initiated the project by immersing myself in the foundational concepts required for the development of a real-time dashboard focused on key process parameters. The primary objective was to understand **object detection techniques**, as they are critical for tracking and analysing process metrics in real time.

**Key Activities:**

1. **Understanding Object Detection**:
   * Researched fundamental concepts behind object detection, including the difference between traditional methods (like Haar cascades) and modern deep learning-based approaches.
   * Explored state-of-the-art algorithms such as **YOLO (You Only Look Once)**, **SSD (Single Shot Detector)**, and **Faster R-CNN**.
   * Evaluated the pros and cons of each method in terms of speed, accuracy, and real-time applicability.
2. **Literature Review & Use Cases**:
   * Reviewed academic papers, blog posts, and documentation to identify best practices and real-world implementations of dashboards for industrial and IoT use cases.
   * Focused on understanding how industries visualize critical KPIs and integrate real-time detection systems with dashboards.
3. **Tool & Framework Evaluation**:
   * Explored various libraries and tools for implementing object detection like **OpenCV**, **TensorFlow**, **PyTorch**, and **MediaPipe**.
   * Investigated dashboard visualization frameworks such as **Plotly Dash**, **Grafana**, **Power BI**, and **custom web dashboards** using **JavaScript (D3.js / Chart.js)**.
4. **Environment Setup**:
   * Set up a working development environment with required libraries and frameworks.
   * Installed and configured relevant Python/JavaScript libraries, Jupyter notebooks, and IDEs to streamline further development.

**Outcome:**

By the end of Week 1, I had gained a clear understanding of the object detection pipeline, shortlisted a few candidate models for implementation, and outlined an initial system architecture. This foundational research set the stage for the practical implementation in the upcoming weeks.

import cv2

import numpy as np

import matplotlib.pyplot as plt

image\_path = "test.png"

image = cv2.imread(image\_path)

if image is None:

    raise FileNotFoundError(f"Image not found: {image\_path}")

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# Histogram equalization to enhance contrast

equalized = cv2.equalizeHist(gray)

# Apply Gaussian Blur

blurred = cv2.GaussianBlur(equalized, (5, 5), 1)

# Canny edge detection (for visualization, optional)

edges = cv2.Canny(blurred, 50, 150)

# Hough Circle detection with tuned parameters

circles = cv2.HoughCircles(

    blurred,

    cv2.HOUGH\_GRADIENT,

    dp=1.0,

    minDist=15,

    param1=70,   # Lower Canny high threshold for edge detection inside Hough

    param2=23,   # Lower accumulator threshold to detect more circles

    minRadius=10,

    maxRadius=35

)

output = image.copy()

if circles is not None:

    circles = np.round(circles[0, :]).astype("int")

    print(f"Circles detected before filtering: {len(circles)}")

    # For now, skip overlap filtering to maximize detections

    for (x, y, r) in circles:

        cv2.circle(output, (x, y), r, (0, 255, 0), 2)

        cv2.rectangle(output, (x - 2, y - 2), (x + 2, y + 2), (0, 0, 255), -1)

else:

    print("No circles detected.")

plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)

plt.title("Equalized + Blurred")

plt.imshow(equalized, cmap='gray')

plt.axis("off")

plt.subplot(1, 3, 2)

plt.title("Canny Edges")

plt.imshow(edges, cmap='gray')

plt.axis("off")

plt.subplot(1, 3, 3)

plt.title("Detected Circles")

plt.imshow(cv2.cvtColor(output, cv2.COLOR\_BGR2RGB))

plt.axis("off")

plt.show()