**Donut Defect Detector: A Report on its Workflow**

**📋 Application Overview**

The **Donut Defect Detector** is a Python application that uses computer vision to assess the quality of donut-shaped objects. It operates by comparing the geometric properties of a test object against a known, defect-free reference object. The application's workflow is divided into a user-facing graphical interface and a backend analysis engine.

**🖥️ GUI Workflow (User Interaction)**

The graphical user interface, built with tkinter, provides a simple, step-by-step process for the user.

1. **Select Reference Image:** The user first clicks a "Browse" button to open a file dialog and select a single image of a perfect, defect-free donut. This image serves as the **"gold standard"** for all subsequent comparisons.
2. **Select Test Image:** Next, the user clicks a similar "Browse" button to select a single image of the donut they wish to test.
3. **Run Analysis:** Once both images have been selected, the "Run Analysis" button is enabled. Clicking this button initiates the core defect detection process.
4. **View Results:** The application's output is displayed in a scrolling text box. It provides real-time updates on the analysis, including whether the test object is classified as **GOOD** or **DEFECTIVE**, and details on the detected defect type and magnitude.

**⚙️ Analysis Workflow (Backend Process)**

The core logic of the application is a two-step process: first, it analyzes the geometric properties of the reference image, and second, it repeats this analysis for the test image to perform a comparison.

**Step 1: Get Donut Parameters**

The get\_donut\_parameters() function is the heart of the analysis. It takes an image as input and returns the object's center, outer radius, and inner radius. The process is as follows:

1. **Image Pre-processing:** The image is converted to grayscale to simplify processing. It then undergoes **adaptive thresholding** to create a clean black-and-white image, effectively separating the donut from the background.
2. **Morphological Operations:** The black-and-white image is cleaned up using **morphological operations** (OPEN and CLOSE). This removes small imperfections and fills minor gaps, ensuring the donut's shape is a single, clean contour.
3. **Contour Detection:** The script finds the main shape in the image by detecting its **contour**. The largest contour is assumed to be the donut itself.
4. **Geometric Scan:** A **360-degree scan** is performed from the calculated center of the donut. At each degree, the script measures the distance from the center to the inner and outer boundaries of the object. It averages these measurements to obtain the final inner and outer radii.

**Step 2: Comparison and Classification**

Once the get\_donut\_parameters() function has been run on both the reference and test images, the results are compared.

1. **Calculate Difference:** The script calculates the absolute difference between the test donut's outer radius and the reference donut's outer radius. It does the same for the inner radii.
2. **Determine Defect:** A **total difference** is calculated by summing the two differences. If this sum exceeds a predefined defect\_threshold (e.g., a few pixels), the object is classified as defective.
3. **Categorize Defect Type:** To provide more detail, the script checks which radius difference is greater. A larger difference in the outer radius indicates an "Extra Portion (blob)," while a larger difference in the inner radius indicates a "Missing Portion (chip)."

This robust geometric approach is effective because it is resilient to minor changes in lighting or position, making it a reliable method for quality control.

**2)** **Report on the 3D Bee Path Plotter**

This document explains the functionality and structure of the Python script designed to plot the path of a dynamical system in three-dimensional space. The code effectively simulates and visualizes the movement of a "bee" following a set of differential equations.

**1. The Core Equations: Modeling the Movement**

At the heart of the program is the bee\_equations function. This function is a mathematical model of the system's behavior. It takes the bee's current position (x, y, z) and the simulation parameters (a, b, c) as input. Its sole job is to calculate the **rate of change** of the bee's position in each dimension (x\_dot, y\_dot, z\_dot) at that exact moment.

The three equations provided by you—x˙=a∗(y−b), y˙=b∗x−y−x∗z, and z˙=x∗y−c∗z—are examples of a set of **ordinary differential equations (ODEs)**. The program translates these into the Python function, which serves as a blueprint for the numerical solver.

**2. Setting up the Simulation**

Before the path can be calculated, the program must define the initial conditions and parameters. This is handled in the main part of the script.

* **Parameters:** The variables a, b, and c are set to 10.0, 28.0, and 2.667 respectively. These are the fixed constants that govern the system's dynamics. Changing these values would fundamentally alter the "bee's" path.
* **Initial Conditions:** The initial position of the bee at time t=0 is set with x0, y0, and z0. These values (0.0, 1.0, and 1.05) are the starting point from which the entire path unfolds.
* **Time Integration:** The np.linspace function creates a time array t that goes from 0 to 50 seconds, with 5000 data points. This array tells the program exactly at which points in time it needs to calculate the bee's position to create a smooth, continuous-looking plot.

**3. Solving the Equations with odeint**

This is where the magic happens. The scipy.integrate.odeint function is a powerful numerical solver. It takes the bee\_equations function, the initial\_state (our starting point), and the t (time array) as its primary inputs.

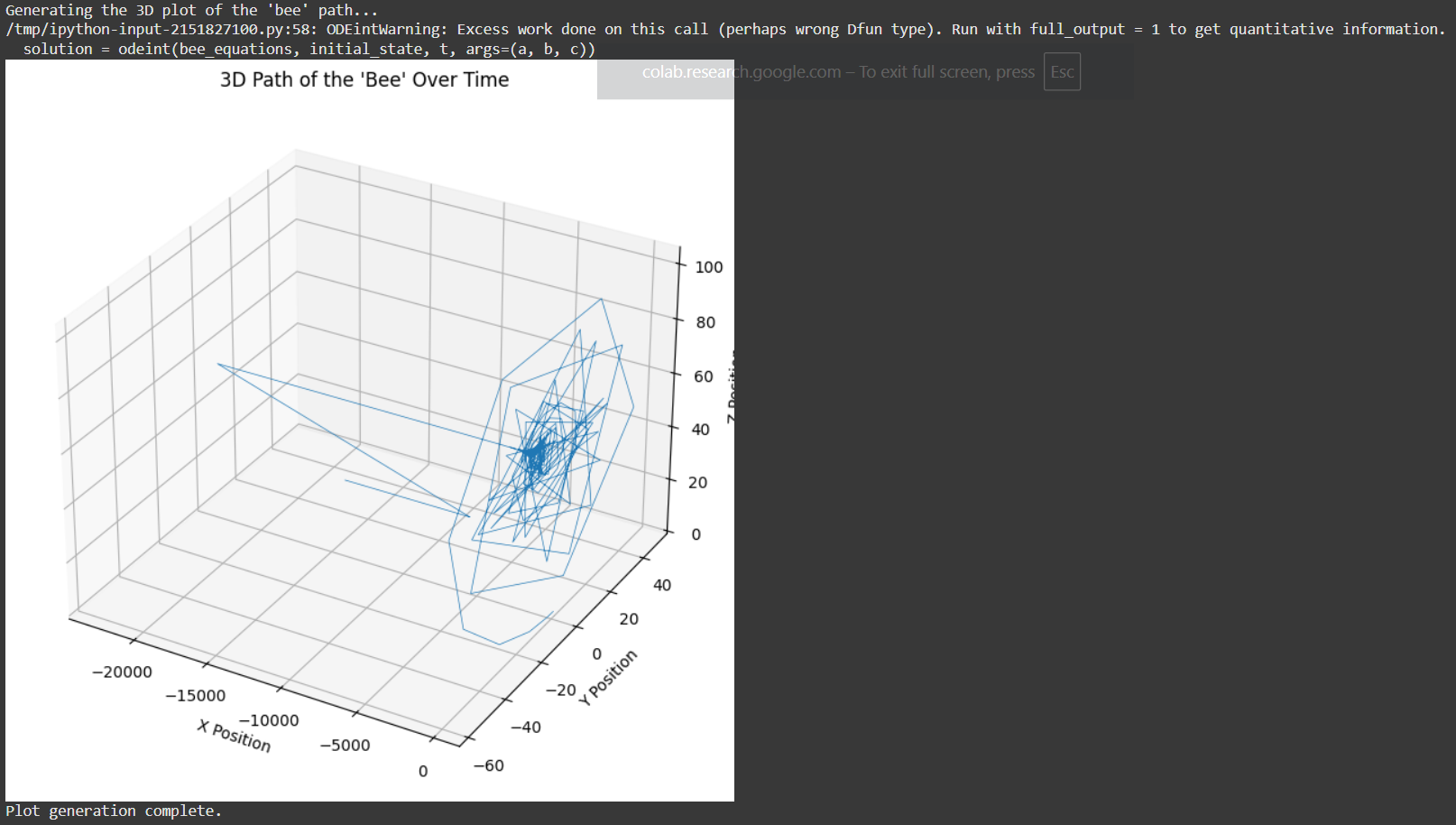
The odeint function works iteratively:

1. It starts at the initial position (x0, y0, z0) at t=0.
2. It uses the bee\_equations function to calculate the velocity vector (x\_dot, y\_dot, z\_dot).
3. It then takes a small step forward in time, using that velocity to estimate the new position.
4. This process is repeated for every point in the t array, producing a complete numerical solution for the bee's position over time.

The result is stored in the solution variable, which is a NumPy array containing the x, y, and z coordinates for each time step.

**4. Plotting the Results in 3D**

The final section of the code is dedicated to visualizing the data using matplotlib. The program first creates a figure and a special 3D subplot. It then uses the ax.plot() function to draw a line graph using the x, y, and z coordinate arrays extracted from the solution. The lw=0.5 parameter makes the line thin for better visibility of the complex path.

Finally, the plot is given a title and labeled axes to ensure the data is clearly presented and easy to understand before being displayed to the user.  
  


**Kaggle Dataset Problem**

Purpose: Prepares your working environment for vehicle detection using a dataset from Kaggle.

Steps:

1. Kaggle API Key:
   * Creates .kaggle directory, moves your kaggle.json (API credentials) there, and sets file permissions.
2. Dataset Download & Unzip:
   * Downloads the “vehicle-detection-dataset” from Kaggle.
   * Unzips it for further processing.
3. Install Dependencies:
   * Installs required Python libraries: ultralytics, matplotlib, seaborn, pandas, xmltodict.

**3.**Data Preparation and Annotation Conversion

Purpose: Converts dataset annotations from XML (PASCAL VOC format) to YOLO format (needed for Ultralytics YOLO training).

Steps:

1. Paths:
   * Defines source directories for training and test data.
2. Class Names:
   * Vehicle classes handled: ['car', 'bus', 'motorcycle', 'truck']
3. XML to YOLO Conversion:
   * convert\_xml\_to\_yolo() reads XML files, extracts object classes and bounding boxes.
   * Converts bounding box (xmin, ymin, xmax, ymax) to YOLO format:
     + [class\_id, x\_center, y\_center, width, height], all relative to image size.
4. Train/Val Split & File Organization:
   * Shuffles images, splits into 80% train, 20% val.
   * Copies images and writes converted labels to /content/yolo\_dataset/… subfolders.
5. Count Classes:
   * Counts how many annotations per class (for data insights).
6. Test Set:
   * Copies over test images (images only, as test labels are usually not available).
7. data.yaml:
   * Creates a config file required for YOLO model training, including:
     + Paths to images for train, val, test.
     + Number of classes and their names.

**4.**YOLOv8 Model Training

Purpose: Trains the YOLOv8 object detection model on your custom vehicle dataset.

Steps:

1. YOLO Setup:
   * Loads the pre-trained “nano” YOLOv8 backbone (yolov8n.pt).
2. Starts Training:
   * Runs .train() method with:
     + data='data.yaml' (configuration & paths)
     + epochs=50 (number of training iterations)
     + imgsz=640 (image resolution)
     + Manages experiment outputs under “vehicle\_detection/yolov8\_custom”.

**5.**Prediction on Test Set

Purpose: Uses the best trained model to run object detection on the test set and save results.

Steps:

1. Load Best Model:
   * Finds best model weights saved after training: …/yolov8\_custom/weights/best.pt
2. Predict:
   * Makes predictions on all test images.
   * Saves images with predicted bounding boxes and label files under vehicle\_detection/predictions.