SASTA OLIPHANT SCIENCE AWARDS

SOUTH AUSTRALIAN SCIENCE TEACHERS ASSOCIATION

Highly Commended

Programming, Apps & Robotics

Year 11 - 12

Ginger Vallance

Heathfield High School

Growth Tracker: Fiji + Calibration Edition

Project Controls

New Project

Open Project

No Project Loaded

Experiment Groups

+ Add

Renam

Del

Target Group: None

Measurement

Calibrate Scale

Scale: 1px = 1px

Image Lab

Tuning Lab

Add Individual Images

Add Folder (Avg)

Export

Save Graph (PNG)

Save Data (CSV)

Wipe All Data

1.0
0.8
0.6
0.4
0.2
0.0

1.0
0.8
0.6
0.4
0.2
0.0

0.0
0.2
0.4
0.6
0.8
1.0

Filename
Timestamp
ExG
Size
Notes

Bulk Edit Notes

Delete Row

GIGABYTE

6/9/2026

Processing and Graphing Plant Growth

A program for bio-data analysis

Ginger Vallance
GINGER.VALLANCE762@SCHOOLS.SA.ED

CONTENTS

AIM

This app was created in response to the time-consuming process of individually processing images manually to graph the rate of growth and the health of plants being used for biofuels. It takes the average particle size and the excess green count of images that have been taken by the experimenter and graphs them over time. In agricultural and scientific spaces, such as farming and horticulturists, plants must be analysed over time, this code provides a fast and convenient way to do this so that data can be analysed without issue.

INSTRUCTIONS

This python program can be run on any device capable of displaying and storing images files (.jpg, .jpeg, .png, .tif, .tiff, .bmp) and data.

Loading the code is simple, just click on the icon of the executable after downloading. To use the program (fig1) for the first time, create a new project before beginning, otherwise an old file can be loaded, both buttons are clearly labelled on the right-side bar. From there new data sets can be created, deleted, and added to using all the side tools. When adding new data, it will tell you when items are being processed, larger image files, can take longer. In the bottom section data can be annotated as well as edited and deleted. Additional features such as saving, renaming, loading, calibrating and the image lab can be accessed and changed from their respective buttons.

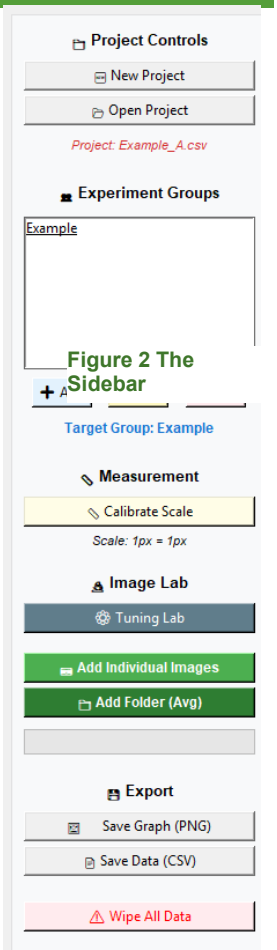
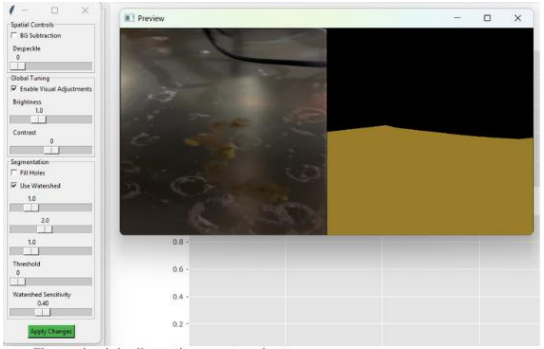


Figure 2 The Sidebar

ANNOTATIONS

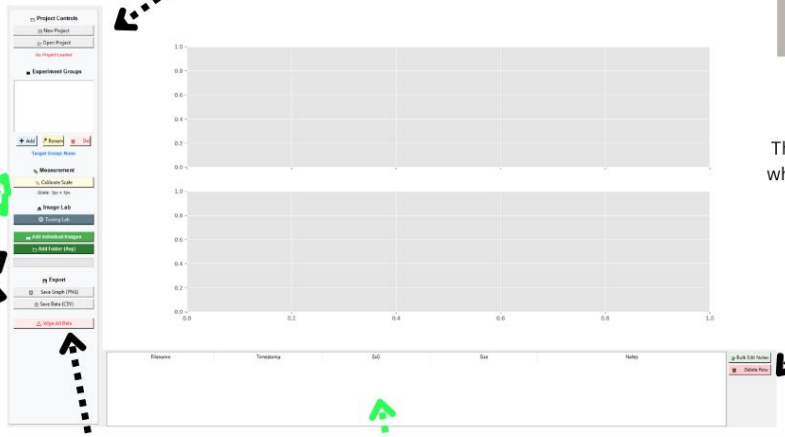


The tuning lab allows the user to select a photo to act as an example when they change the settings, providing a live preview to their changes. It shows all the different settings that go into getting the Growth and Health, except for the measurements (which is in calibration) and the excess green count (which is automatically done in the code by separating the red green and blue colour channels and then adding them together so that there is two green and negetive red and blue)

The New project button allows the user to create a new excel file with all of the information categories empty. The load project button recals these excel files when selected

Key points when using the program:

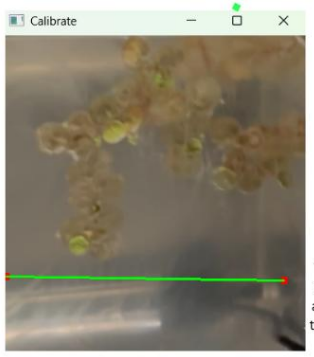
1. To begin you MUST have a project selected, this can be loaded or created in the side bar under the “Program controls section”
2. To add data you have to select a group to add it to, to do this you can select an exsisting group from the “Experiment Group Section” or creat a new group using the blue button under the selection box. These groups can be deleted or renamed using the red and yellow bettons next to it.
3. The image lab changes the parts defined as particles, while the calibration defines how big a pixel is. Both require an image to be selected to demonstrate visually what is happening.
4. Graphs can be exported as pngs and data can be exported as CSV using their respective buttons
5. All data can be wiped using the wipe data button.
6. Individual data points can be edited by double clicking them



The export section allows the user to either save a png of the graphs or save the data in a CVS file.

The red button allows the user to delete a single data point while the green button allows them to add notes to multiple data sets at the same time.

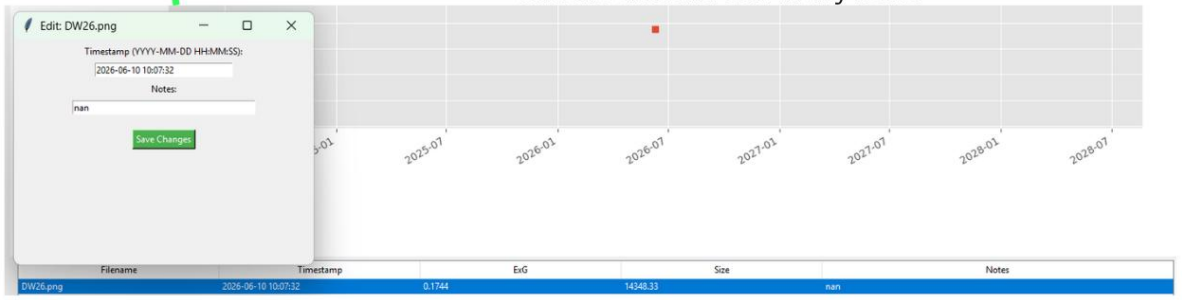
The light green add individual image button takes the growth (average particle size) and the health (excess green count) of a single image selected by the user. The dark green button takes the average of the growth and the health of all images in a folder selected by the user.



This is the calibration screen, it allows the user to select an image from their files, select two points and after clicking enter they can say how far apart they are, so the program can find the size of a pixel in relation to the images.

This wipes all data from a group, resetting it completely.

Once a data point is added to the project it can be double clicked from where the arrow is pointing to edit the date and add in any notes



SCIENTIFIC EXPLANATION AND DEMONSTRATION

There are two things measured in this code, the excess green count, representing the health of the plant, and the average particle size, representing the growth of the plant. The excess green count is a colour index commonly used to assess plant health by looking at the green light reflected in comparison to red and blue light (Reid et al. 2016). This is effective because the chlorophyll in healthy plants reflects green, while the damaged and dying spots are typically shades of brown, due to the lack of chlorophyll, causing these spots to reflect as shades of red (Moore 2018). This is found by separating the different colour channels, overlaying 2 green channels and then removing a red and blue channel from the 2 green channels.

Figure 3 Demonstration graph (18/3-17/4)

Figure 2 shows a graph made using the program for an experiment testing different water plants for there use in biofuels, it demonstrates the growth of water lettuce (purple), red root floaters (blue) and duckweed (red). This graph will act as a demonstration on how to use the program to analysis data with scientific backing.

By looking at the bottom graph demonstrating the health, the downward trend of the plants demonstrates a lowering excess green count. This indicates the plants are losing chlorophyll and subsequently they are dying off. This is because the plants did not have the proper fertiliser and were not getting enough nitrogen. Once the proper fertilizer was bought, according to the notes in the program, the plants experienced a boost in growth and health. While this quick burst of energy died down quickly, the graph in figure 3 demonstrates that this did have an extended positive effect on the plants. This data could be used in the biofuel experiment to mention the importance of fertilizer on plant growth and survival.

The average particle size is used to find the size or spread of leaves and increases or decreases over time. The average leaf size is a measure determining health and growth as increased size indicates larger, healthier plants, it is commonly used when assessing the growth of duckweed and pond plants, the study of which inspired this program (Saurabh Sengupta, Chiranjeeb Medda, Anjana Dewanji 2010). This is found in the code by first applying the excess green count ‘filter’ and then applying the image lab settings, from there the individual particles can be found by either water shedding or contour finding. Once the particles are found, their area is calculated in pixels then converted according to the calibration settings. Finally, the average of all these particles is found.

Code and Explanation

Imports

The following are the imports, they hold information the code uses to function, this includes libraries for graphing and time as well as creating a user interface. Numpy and Pandas is for the image handling and analysis, matplotlib is for the graphing and tkinter is for the user interface.

```
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
import tkinter as tk
from tkinter import filedialog, messagebox, simpledialog, ttk
from datetime import datetime
import os
import threading
```

Dashboard setup and Button Function

Definitions and classes

This 'class' or section of code holds information on each part of the program and how it works, including the way the interface looks what happens when you click on different sections.

```
class ExperimentDashboard:
```

This definition sections sets out the original state of the program when loaded, including the way everything looks and the default settings of the app.

```
    def __init__(self, root):
```

This sets out the "root geometry" which is the original size of the window as well as titling the program when opened.

```
        self.root = root
        self.root.title("Growth Tracker")
        self.root.geometry("1450x950")
```

This sets out the program to have empty graphs and data sets when the program is booted.

```
        self.db_path = None
```

This sets the pixel ratios up so that the screen stays consistent when in different sized windows.

```
        self.pixel_to_unit_ratio = 1.0
        self.unit_name = "px"
```

Settings

These are the "default settings" for the image processing, these include automatically finding the excess green index, applying watershed, not subtracting the background, not filling holes, etc.

```
        self.DEFAULT_SETTINGS = {
            "red_w": 1.0, "green_w": 2.0, "blue_w": 1.0,
            "threshold": 0, "watershed_dist": 0.4,
            "brightness": 1.0, "contrast": 0,
            "use_visuals": True, "use_watershed": True,
            "despeckle": 0, "bg_subtraction": False, "fill_holes": False
        }
```

This sets the data sets when opening the code, it defines that there are currently none active or there.

```
        self.settings = self.DEFAULT_SETTINGS.copy()
        self.all_groups = []
        self.active_set_name = None
```

SIDE BAR SETUP

This sets the “side bar” where all of the controls are, it defines the colour and the size.

```
        self.sidebar = tk.Frame(root, width=280, bg="#f8f9fa", borderwidth=1, relief="ridge")
        self.sidebar.pack(side="left", fill="y", padx=10, pady=10)
```

This sets all of the user interface, it adds in buttons, labels, frames, and canvases to create different features such as lists. It defines several aspects about them including their size, colour, name, font and number of pixels that surround them (to prevent the user from clicking on the wrong button by accident)

```
        tk.Label(self.sidebar, text="🗀 Project Controls", font=("Arial", 10, "bold"),
bg="#f8f9fa").pack(pady=(10,5))
        tk.Button(self.sidebar, text="🆕 New Project", command=self.create_new_project).pack(fill="x",
padx=10, pady=2)
        tk.Button(self.sidebar, text="🗁 Open Project", command=self.switch_project).pack(fill="x", padx=10,
pady=2)

        self.status_var = tk.StringVar(value="No Project Loaded")
        tk.Label(self.sidebar, textvariable=self.status_var, font=("Arial", 8, "italic"), wraplength=250,
bg="#f8f9fa", fg="#d32f2f").pack(pady=5)

        tk.Label(self.sidebar, text="👥 Experiment Groups", font=("Arial", 10, "bold"),
bg="#f8f9fa").pack(pady=(15,5))
        self.group_listbox = tk.Listbox(self.sidebar, height=8, exportselection=False,
selectbackground="#2196f3")
        self.group_listbox.pack(fill="x", padx=10, pady=5)
        self.group_listbox.bind('<<ListboxSelect>>', self.on_group_select)

        grp_btn_frame = tk.Frame(self.sidebar, bg="#f8f9fa")
        grp_btn_frame.pack(fill="x", padx=10)
        tk.Button(grp_btn_frame, text="➕ Add", command=self.add_group_to_list, bg="#e3f2fd",
width=6).pack(side="left", expand=True, pady=2)
        tk.Button(grp_btn_frame, text="🖉 Rename", command=self.rename_group, bg="#fff9c4",
width=6).pack(side="left", expand=True, pady=2)
        tk.Button(grp_btn_frame, text="🗑 Del", command=self.delete_group, bg="#ffebee", fg="#c62828",
width=6).pack(side="left", expand=True, pady=2)

        self.group_summary_label = tk.Label(self.sidebar, text="Target Group: None", font=("Arial", 9,
"bold"), bg="#f8f9fa", fg="#1976d2")
        self.group_summary_label.pack(pady=5)

        tk.Label(self.sidebar, text="📏 Measurement", font=("Arial", 10, "bold"),
bg="#f8f9fa").pack(pady=(15,5))
        tk.Button(self.sidebar, text="📏 Calibrate Scale", command=self.calibrate_scale,
bg="#fffde7").pack(fill="x", padx=10, pady=2)
        self.scale_label = tk.Label(self.sidebar, text="Scale: 1px = 1px", font=("Arial", 8, "italic"),
bg="#f8f9fa")
        self.scale_label.pack()

        tk.Label(self.sidebar, text="🔬 Image Lab", font=("Arial", 10, "bold"),
bg="#f8f9fa").pack(pady=(15,5))
        tk.Button(self.sidebar, text="⚙ Tuning Lab", command=self.open_tuning_lab, bg="#607d8b",
fg="white").pack(fill="x", padx=10, pady=2)

        tk.Button(self.sidebar, text="📷 Add Individual Images", command=self.add_data_point, bg="#4caf50",
fg="white", font=("Arial", 9, "bold")).pack(fill="x", padx=10, pady=(15,2))
        tk.Button(self.sidebar, text="🗀 Add Folder (Avg)", command=self.add_folder_avg, bg="#2e7d32",
fg="white", font=("Arial", 9, "bold")).pack(fill="x", padx=10, pady=(2,5))

        self.progress = ttk.Progressbar(self.sidebar, orient="horizontal", length=200, mode="determinate")
        self.progress.pack(fill="x", padx=10, pady=5)

        tk.Label(self.sidebar, text="💾 Export", font=("Arial", 10, "bold"), bg="#f8f9fa").pack(pady=(15,5))
        tk.Button(self.sidebar, text="🖼 Save Graph (PNG)", command=self.save_graph_image).pack(fill="x",
padx=10, pady=2)
        tk.Button(self.sidebar, text="🗎 Save Data (CSV)", command=self.export_csv).pack(fill="x", padx=10,
pady=2)
        tk.Button(self.sidebar, text="⚠ Wipe All Data", command=self.clear_all_project_data, fg="red",
bg="#ffebee").pack(fill="x", padx=10, pady=20)

        self.right_container = tk.Frame(root)
        self.right_container.pack(side="right", fill="both", expand=True)
```

GRAPH AND DATA SET SETUP

This begins setting up the graphs including their size, labels, surrounding pixels and axis.

```
        plt.style.use('ggplot')
        self.fig, (self.ax1, self.ax2) = plt.subplots(2, 1, figsize=(8, 5), sharex=True)
        self.canvas = FigureCanvasTkAgg(self.fig, master=self.right_container)
        self.canvas.get_tk_widget().pack(side="top", fill="both", expand=True)

        self.table_frame = tk.Frame(self.right_container)
        self.table_frame.pack(side="bottom", fill="x", padx=10, pady=5)
```

This begins to set up the bottom bar where the data point information is stored, it sets the size and the labels, as well as the rows. It also defines that something will happen when one of the items is double clicked.

```
        cols = ("Filename", "Timestamp", "ExG", "Size", "Notes")
        self.tree = ttk.Treeview(self.table_frame, columns=cols, show='headings', height=8)
        for c in cols: self.tree.heading(c, text=c)
        self.tree.column("Filename", width=200)
        self.tree.column("Notes", width=400)
        self.tree.pack(side="left", fill="both", expand=True)
        self.tree.bind("<Double-1>", self.on_row_double_click)
```

This sets up the bottom buttons.

```
        table_btn_frame = tk.Frame(self.table_frame)
        table_btn_frame.pack(side="right", fill="y", padx=5)
        tk.Button(table_btn_frame,        text="        Bulk        Edit        Notes",        bg="#e8f5e9",
command=self.bulk_edit_notes).pack(fill="x", pady=2)
        tk.Button(table_btn_frame,          text="          Delete          Row",          bg="#ffcdd2",
command=self.delete_selected_row).pack(fill="x", pady=2)
```

IMAGE ANALYSIS

This defines what happens when an image is being analysed, it applies all the settings from the image lab, including the brightness, contrast, excess green index, thresholds, masks watershed, etc.

```
    def run_analysis(self, img, s):
        work_img = img.copy()
```

This is the brightness and contrast.

```
        if    s.get("use_visuals"):    work_img    =    cv2.convertScaleAbs(work_img,    alpha=s["brightness"],
beta=s["contrast"])
```

This is blur and despeckling as well as background subtraction.

```
        if s.get("despeckle", 0) > 0: work_img = cv2.medianBlur(work_img, (s["despeckle"]*2)+1)
        if s.get("bg_subtraction"):
            work_img              =              cv2.morphologyEx(work_img,              cv2.MORPH_TOPHAT,
cv2.getStructuringElement(cv2.MORPH_RECT, (21,21)))
```

This is the excess green index math, threshold and fill holes.

```
        b, g, r = cv2.split(work_img.astype(float))
        denom = np.clip(r+g+b, 1, None)
        exg = (s["green_w"]*(g/denom)) - (s["red_w"]*(r/denom)) - (s["blue_w"]*(b/denom))
        e8 = cv2.normalize(exg, None, 0, 255, cv2.NORM_MINMAX).astype(np.uint8)
        _,   th   =   cv2.threshold(e8,   s["threshold"],   255,   cv2.THRESH_BINARY   +   (cv2.THRESH_OTSU   if
s["threshold"]==0 else 0))
        if s.get("fill_holes"): th = cv2.morphologyEx(th, cv2.MORPH_CLOSE, np.ones((5,5)))
```

This is watershed, masking, and distancing.

```
        mask = np.zeros_like(work_img)
        px_areas = []
        if s.get("use_watershed"):
            dist = cv2.distanceTransform(cv2.morphologyEx(th, cv2.MORPH_OPEN, np.ones((3,3))), cv2.DIST_L2,
5)
            if dist.max() > 0:
                _, fg = cv2.threshold(dist, s["watershed_dist"] * dist.max(), 255, 0)
                _, mk = cv2.connectedComponents(np.uint8(fg))
                mk = cv2.watershed(work_img, mk + 1)
                px_areas = [np.sum(mk == l) for l in np.unique(mk) if l > 1]
                for l in np.unique(mk):
                    if l > 1: mask[mk == l] = [np.random.randint(0,255) for _ in range(3)]
        else:
            cnts, _ = cv2.findContours(th, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

```
            px_areas = [cv2.contourArea(c) for c in cnts if cv2.contourArea(c) > 5]
            cv2.drawContours(mask, cnts, -1, (0, 255, 0), -1)
```

This is applying the pixel to unit ratio that can be edited within the calibration.

```
        real_areas = [a * (self.pixel_to_unit_ratio**2) for a in px_areas]
        return np.mean(exg), len(real_areas), np.mean(real_areas) if real_areas else 0, mask
```

ADDING NEW DATA

This defines the process for adding in different data points.

```
    def add_data_point(self):
This states that if there is no group selected, to show a popup that tells the user that.
        if not self.db_path or not self.active_set_name:
            messagebox.showwarning("Warning", "Please select a Targeted Group first!")
            return
```

This sets the progress bar and selects the image, it uses threading so not to crash the program by overloading the system

```
        paths = filedialog.askopenfilenames()
        if not paths: return
        self.progress['value'] = 0
        self.progress['maximum'] = len(paths)
        self.status_var.set(f"⏳ Processing 0/{len(paths)}")
        threading.Thread(target=self._process_images_worker, args=(paths,), daemon=True).start()
```

This defines the process of getting the data that has been collected and saved and graphing it at the proper times and places on the graph. It sets the name based on the file name of the image or folder and defines the time and date as the current time and date.

```
    def _process_images_worker(self, paths):
        d = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        for i, p in enumerate(paths):
            img = cv2.imread(p)
            if img is None: continue
            fname = os.path.basename(p)
            ev, count, sv, mask = self.run_analysis(img, self.settings)
            row   =   {"Timestamp":   d,   "Set_ID":   self.active_set_name,   "Avg_ExG":   round(ev,4),
"Avg_Particle_Size": round(sv,2), "Notes": "", "Filename": fname}
            pd.DataFrame([row]).to_csv(self.db_path, mode='a', header=False, index=False)
```

This sets the text for how many images have been processed.

```
            self.root.after(0, lambda x=i+1, t=len(paths): [self.progress.step(1), self.status_var.set(f"⏳
Processing {x}/{t}")])
```

This finishes the processing bar.

```
        Self.root.after(0, self._finish_processing)
```

This defines how to take the average data from a folder and graphs it. It works very similar to the individual image adding, but it has different progress bar texts and also does the math for finding the average values of each of the images. It also sends a popup to the user if it cannot find any of the valid file formats within the folder.

```
    def add_folder_avg(self):
        if not self.db_path or not self.active_set_name:
            messagebox.showwarning("Warning", "Please select a Targeted Group first!")
            return
        folder = filedialog.askdirectory()
        if not folder: return
        self.progress['value'] = 0
        self.status_var.set("⏳ Searching Folder...")
        threading.Thread(target=self._process_folder_worker, args=(folder,), daemon=True).start()

    def _process_folder_worker(self, folder):
        valid_exts = ('.jpg', '.jpeg', '.png', '.tif', '.tiff', '.bmp')
        paths = [os.path.join(folder, f) for f in os.listdir(folder) if f.lower().endswith(valid_exts)]

        if not paths:
            self.root.after(0, lambda: messagebox.showerror("Error", "No valid images found in folder."))
            return
        self.root.after(0, lambda: self.progress.config(maximum=len(paths)))
        exg_values, size_values = [], []
        d = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
```

```
        for i, p in enumerate(paths):
            img = cv2.imread(p)
            if img is None: continue
            ev, count, sv, _ = self.run_analysis(img, self.settings)
            exg_values.append(ev)
            size_values.append(sv)
            self.root.after(0, lambda x=i+1, t=len(paths): [self.progress.step(1), self.status_var.set(f"⌛ Analyzed {x}/{t}")])

        if exg_values:
            final_exg = np.mean(exg_values)
            final_size = np.mean(size_values)
            folder_name = f"Folder: {os.path.basename(folder)}"
            row = {"Timestamp": d, "Set_ID": self.active_set_name, "Avg_ExG": round(final_exg,4), "Avg_Particle_Size": round(final_size,2), "Notes": f"Avg of {len(paths)} images", "Filename": folder_name}
            pd.DataFrame([row]).to_csv(self.db_path, mode='a', header=False, index=False)

        self.root.after(0, self._finish_processing)
```

This tells the user when the images have been processed and makes sure the user interface is up to date.

```
    def _finish_processing(self):
        self.progress['value'] = 0
        self.status_var.set(f"Project: {os.path.basename(self.db_path)}")
        self.refresh_ui()
        messagebox.showinfo("Success", "Data processing complete!")
```

This makes refreshes the areas where data is displayed, including the graph and data sets. It checks each section for data and makes sure it is displayed properly.

```
    def refresh_ui(self):
        self.group_listbox.delete(0, tk.END)
        for g in self.all_groups: self.group_listbox.insert(tk.END, g)

        df = pd.DataFrame()
        if self.db_path and os.path.exists(self.db_path):
            df = pd.read_csv(self.db_path)
            if not df.empty:
                df['Timestamp'] = pd.to_datetime(df['Timestamp'])

        self.ax1.clear(); self.ax2.clear()
        if not df.empty:
            df = df.sort_values('Timestamp')
            for lbl, grp in df.groupby('Set_ID'):
                self.ax1.plot(grp['Timestamp'], grp['Avg_Particle_Size'], marker='o', label=lbl)
                self.ax2.plot(grp['Timestamp'], grp['Avg_ExG'], marker='s', linestyle='--', label=lbl)
            self.ax1.legend(loc='upper left', fontsize='x-small')
            self.fig.autofmt_xdate()

        self.ax1.set_title(f"Growth ({self.unit_name}^2)"); self.ax2.set_title("Health (ExG)"); self.canvas.draw()

        for i in self.tree.get_children(): self.tree.delete(i)
        if not df.empty:
            disp = df[df['Set_ID'] == self.active_set_name] if self.active_set_name else df
            for idx, row in disp.iterrows():
                self.tree.insert("", "end", values=(row.get('Filename', "N/A"), row['Timestamp'], row['Avg_ExG'], row['Avg_Particle_Size'], row.get('Notes', "")))
```

EDITING AND DELETING DATA

This defines what happens when a row or data point is deleted.

```
    def delete_selected_row(self):
        sel = self.tree.selection()
        if not sel: return
        vals = self.tree.item(sel[0])['values']
        f_name, ts = vals[0], str(vals[1])
        if messagebox.askyesno("Confirm", f"Delete record for {f_name}?"):
            df = pd.read_csv(self.db_path)
            mask = (df['Filename'] == f_name) & (df['Timestamp'] == ts) & (df['Set_ID'] == self.active_set_name)
            df = df[~mask]
            df.to_csv(self.db_path, index=False)
            self.refresh_ui()
```

This is what happens when the data points are clicked on twice, it allows the user to edit the information through a popup and saves it, as well as making sure it is on the graph.

```
    def on_row_double_click(self, event):
        sel = self.tree.selection()
        if not sel: return
```

```
        val = self.tree.item(sel[0], "values")
        f_name_orig, ts_orig = val[0], val[1]

        ew = tk.Toplevel(self.root)
        ew.title(f"Edit: {f_name_orig}")
        ew.geometry("400x300")

        tk.Label(ew, text="Timestamp (YYYY-MM-DD HH:MM:SS):").pack(pady=5)
        t_ent = tk.Entry(ew, width=30); t_ent.insert(0, ts_orig); t_ent.pack()
        tk.Label(ew, text="Notes:").pack(pady=5)
        n_ent = tk.Entry(ew, width=40); n_ent.insert(0, val[4]); n_ent.pack()
```

SAVING DATA AND NOTES

This defines what happens when the files are saved by putting the data from the graph into a csv file.

```
        def save_changes():
            try:
                datetime.strptime(t_ent.get(), "%Y-%m-%d %H:%M:%S")
                df = pd.read_csv(self.db_path)
                mask = (df['Filename'] == f_name_orig) & (df['Timestamp'] == ts_orig) & (df['Set_ID'] ==
self.active_set_name)
                df.loc[mask, "Timestamp"] = t_ent.get()
                df.loc[mask, "Notes"] = n_ent.get()
                df.to_csv(self.db_path, index=False)
                ew.destroy(); self.refresh_ui()
            except: messagebox.showerror("Error", "Invalid Format")
        tk.Button(ew, text="Save Changes", bg="#4caf50", fg="white", command=save_changes).pack(pady=20)
```

This allows the user to edit the notes of several data points at the same time.

```
    def bulk_edit_notes(self):
        if not self.active_set_name: return
        new_note = simpledialog.askstring("Bulk Edit", f"Apply note to ALL in '{self.active_set_name}':")
        if new_note is not None:
            df = pd.read_csv(self.db_path)
            df.loc[df['Set_ID'] == self.active_set_name, "Notes"] = new_note
            df.to_csv(self.db_path, index=False); self.refresh_ui()
```

This allows the user to create a new project by saving it to a folder as a csv file with a name chosen by the user, it then updates the graph to be new.

```
    def create_new_project(self):
        p = filedialog.asksaveasfilename(defaultextension=".csv")
        if p:
            pd.DataFrame(columns=["Timestamp",    "Set_ID",    "Avg_ExG",    "Avg_Particle_Size",    "Notes",
"Filename"]).to_csv(p, index=False)
            self.db_path = p; self.status_var.set(f"Project: {os.path.basename(p)}"); self.refresh_ui()
This allows the user to open a csv file and applies it to the graphs.
    def switch_project(self):
        p = filedialog.askopenfilename(filetypes=[("CSV Files", "*.csv")])
        if p:
            self.db_path = p; self.status_var.set(f"Project: {os.path.basename(p)}")
            try: self.all_groups = sorted(pd.read_csv(p)['Set_ID'].unique().tolist())
            except: self.all_groups = []
            self.refresh_ui()
```

DATA GROUPS AND NAMING

This defines the group that the user currently has selected as the target group and displays the data points of that group below, as well as adding new data to that group.

```
    def on_group_select(self, e):
        sel = self.group_listbox.curselection()
        if sel:
            self.active_set_name = self.group_listbox.get(sel[0])
            self.group_summary_label.config(text=f"Target Group: {self.active_set_name}")
            self.refresh_ui()
```

This adds a new group of data to the list and lets the user select a name.

```
    def add_group_to_list(self):
        n = simpledialog.askstring("New Group", "Group Name:")
        if n and n not in self.all_groups:
            self.all_groups.append(n); self.all_groups.sort(); self.refresh_ui()
```

This defines the process for renaming current groups and changing it within the csv file all of the data gets uploaded to.

```
    def rename_group(self):
        sel = self.group_listbox.curselection()
        if not sel: return
        old_name = self.group_listbox.get(sel[0])
        new_name = simpledialog.askstring("Rename", f"Rename '{old_name}' to:")
        if new_name and new_name != old_name:
            if self.db_path and os.path.exists(self.db_path):
                df = pd.read_csv(self.db_path)
                df.loc[df['Set_ID'] == old_name, 'Set_ID'] = new_name
                df.to_csv(self.db_path, index=False)
            idx = self.all_groups.index(old_name)
            self.all_groups[idx] = new_name
            self.all_groups.sort(); self.active_set_name = new_name; self.refresh_ui()
```

This defines what happens when the user deletes a group, it wipes all of the data, after confirming.

```
    def delete_group(self):
        sel = self.group_listbox.curselection()
        if not sel: return
        target = self.group_listbox.get(sel[0])
        if messagebox.askyesno("Confirm", f"Delete group '{target}'?"):
            if self.db_path and os.path.exists(self.db_path):
                df = pd.read_csv(self.db_path)
                df = df[df['Set_ID'] != target]
                df.to_csv(self.db_path, index=False)
            self.all_groups.remove(target); self.active_set_name = None; self.refresh_ui()
```

CALIBRATION AND IMAGE LAB

This calibrates the scale by asking the user to select two points on a chosen image and selecting how many units it is, and what units that measurement is in, checking to make sure the number and unit is real before applying it.

```
    def calibrate_scale(self):
        path = filedialog.askopenfilename()
        if not path: return
        img = cv2.imread(path); clone = img.copy(); pts = []
        def click(event, x, y, f, p):
            if event == cv2.EVENT_LBUTTONDOWN:
                pts.append((x, y)); cv2.circle(clone, (x, y), 5, (0, 0, 255), -1)
                if len(pts) == 2: cv2.line(clone, pts[0], pts[1], (0, 255, 0), 2)
                cv2.imshow("Calibrate", clone)
        cv2.imshow("Calibrate",    clone);    cv2.setMouseCallback("Calibrate",    click);    cv2.waitKey(0);
cv2.destroyAllWindows()
        if len(pts) == 2:
            px = np.sqrt((pts[1][0]-pts[0][0])**2 + (pts[1][1]-pts[0][1])**2)
            real = simpledialog.askfloat("Scale", "Real distance:")
            unit = simpledialog.askstring("Scale", "Unit:")
            if real and unit:
                self.pixel_to_unit_ratio = real / px; self.unit_name = unit
                self.scale_label.config(text=f"Scale: 1px = {self.pixel_to_unit_ratio:.4f} {unit}")
```

This defines the ‘tuning lab’ where the user can change the way the images are analysed. It opens a series of sliders and saves the information to the CSV file. It also displays an image and the way the settings effect the image. The default settings can also be returned to.

```
    def open_tuning_lab(self):
        path = filedialog.askopenfilename()
        if not path: return
        img = cv2.imread(path)
        win = tk.Toplevel(self.root); win.title("Tuning Lab")
        v_vis,    v_wat,    v_bg,    v_fill    =    [tk.BooleanVar(value=self.settings[k])    for    k    in
["use_visuals","use_watershed","bg_subtraction","fill_holes"]]
        def update_p(*args):
            self.settings.update({"red_w":s_r.get(),        "green_w":s_g.get(),        "blue_w":s_b.get(),
"threshold":s_t.get(),    "watershed_dist":s_w.get(),    "brightness":s_br.get(),    "contrast":s_co.get(),
"use_visuals":v_vis.get(),             "use_watershed":v_wat.get(),             "bg_subtraction":v_bg.get(),
"fill_holes":v_fill.get(), "despeckle":s_dp.get()})
            _, _, _, m = self.run_analysis(img, self.settings)
            cv2.imshow("Preview", np.hstack((cv2.resize(img,(400,400)), cv2.resize(m,(400,400)))))
        f1 = tk.LabelFrame(win, text="Spatial Controls"); f1.pack(fill="x", padx=10)
        tk.Checkbutton(f1, text="BG Subtraction", variable=v_bg, command=update_p).pack(anchor="w")
        s_dp  =  tk.Scale(f1,  from_=0,  to=5,  label="Despeckle",  orient="horizontal",  command=update_p);
s_dp.set(self.settings["despeckle"]); s_dp.pack(fill="x")
        f2 = tk.LabelFrame(win, text="Global Tuning"); f2.pack(fill="x", padx=10)
        tk.Checkbutton(f2,         text="Enable         Visual         Adjustments",         variable=v_vis,
command=update_p).pack(anchor="w")
        s_br  =  tk.Scale(f2,  from_=0.1,  to=3.0,  resolution=0.1,  label="Brightness",  orient="horizontal",
command=update_p); s_br.set(self.settings["brightness"]); s_br.pack(fill="x")
        s_co  =  tk.Scale(f2,  from_=-100,  to=100,  label="Contrast",  orient="horizontal",  command=update_p);
s_co.set(self.settings["contrast"]); s_co.pack(fill="x")
        f3 = tk.LabelFrame(win, text="Segmentation"); f3.pack(fill="x", padx=10)
```

```
        tk.Checkbutton(f3, text="Fill Holes", variable=v_fill, command=update_p).pack(anchor="w")
        tk.Checkbutton(f3, text="Use Watershed", variable=v_wat, command=update_p).pack(anchor="w")
        s_r, s_g, s_b = [tk.Scale(f3, from_=0, to=5, resolution=0.1, orient="horizontal", command=update_p)
for _ in range(3)]
        s_r.set(self.settings["red_w"]); s_g.set(self.settings["green_w"]); s_b.set(self.settings["blue_w"])
        s_r.pack(fill="x"); s_g.pack(fill="x"); s_b.pack(fill="x")
        s_t = tk.Scale(f3, from_=0, to=255, label="Threshold", orient="horizontal", command=update_p);
s_t.set(self.settings["threshold"]); s_t.pack(fill="x")
        s_w = tk.Scale(f3, from_=0.1, to=0.9, resolution=0.05, label="Watershed Sensitivity",
orient="horizontal", command=update_p); s_w.set(self.settings["watershed_dist"]); s_w.pack(fill="x")
        tk.Button(win, text="Apply Changes", bg="#4caf50", command=lambda: [win.destroy(),
cv2.destroyAllWindows()]).pack(pady=10)
This defines how to save the graph as a png file by asking the user for a name and saving the figures.
    def save_graph_image(self):
        p = filedialog.asksaveasfilename(defaultextension=".png"); self.fig.savefig(p, dpi=300) if p else
None
This defines how it is saved as an csv file by asking the user for a name then saving all of the data.
    def export_csv(self):
        p = filedialog.asksaveasfilename(defaultextension=".csv")
        if p and self.db_path: pd.read_csv(self.db_path).to_csv(p,index=False)
```

DATA WIPING

This defines what happens when the user wipes all data, the program asks for confirmation before removing all data.

```
    def clear_all_project_data(self):
if self.db_path and messagebox.askyesno("Confirm", "Wipe ALL data?"):
pd.DataFrame(columns=["Timestamp", "Set_ID", "Avg_ExG", "Avg_Particle_Size", "Notes",
"Filename"]).to_csv(self.db_path, index=False); self.refresh_ui()
```

This runs the code when opened.

```
if __name__ == "__main__"
root = tk.Tk(); app = ExperimentDashboard(root); root.mainloop()
```

CONCLUSION

This program's scientific purpose is to analysis the growth and health of biomatter. It is targeted at researchers aiming to study plants, such as farmers and horticulturists. It has been applied on a small-scale research experiment studying the growth of the pond plants, water lettuce, red root floaters, duckweed and azolla, in order to analysis their potential use as biofuels.

BIBLIOGRAPHY

Reid, AM, Chapman, WK, Prescott, CE & Nijland, W 2016, 'Using excess greenness and green chromatic coordinate colour indices from aerial images to assess lodgepole pine vigour, mortality and disease occurrence', *Forest Ecology and Management*, vol. 374, Elsevier BV, pp. 146–153, viewed 10 June 2026, <https://www.sciencedirect.com/science/article/abs/pii/S0378112716302298>.

Moore, G 2018, *Curious Kids: Why are leaves green?*, Find an Expert : The University of Melbourne, The Conversation, viewed 10 June 2026, <https://findanexpert.unimelb.edu.au/news/5766-curious-kids--why-are-leaves-green%3F>.

Sengupta, Saurabh, et al. "The Impact of Duckweed Growth on Water Quality in Sub-Tropical Ponds." *The Environmentalist*, vol. 30, no. 4, Dec. 2010, pp. 353–360, https://doi.org/10.1007/s10669-010-9293-6.