Py_to_PDF

May 8, 2025

[ ]:
```python
#!/usr/bin/env python3
"""
Water-line detection module.

This script processes a single image to detect the waterline using either
Kolmogorov-Smirnov (KS) or mean-difference metrics across horizontal
bands. It produces visual outputs (crop box overlay, 2×4 mode overview)
and logs results to CSV and a USB drive if mounted.

Key components:
- CSV logging helper
- Image rotation, cropping, and visualization
- Difference metrics (mean vs KS) and smoothing
- Peak detection for waterline localization
- 2×4 visualization per mode (original, HSV value, hue, saturation)
- Main process_image() function tying it all together

Configuration is supplied via wd__config_cycle.CONFIG.
"""

import os
import csv
import logging
import shutil

import numpy as np
from PIL import Image

import matplotlib
matplotlib.use("Agg")  # Use non-GUI backend for file output
import matplotlib.pyplot as plt
from matplotlib.colors import rgb_to_hsv
from matplotlib.patches import Rectangle
from scipy.ndimage import gaussian_filter1d
from scipy.signal import find_peaks
from scipy.stats import ks_2samp
```

```python
# Set up module-level logger
logger = logging.getLogger(__name__)

# External configuration dictionary
from wd__config_cycle import CONFIG

# Paths for CSV output depending on system
CSV_FILE_PI = "/home/bjorn/Desktop/wd__directory/wd__results/algorithm_results.
↪csv"
CSV_FILE_PC = r"C:\\Users\\bjorn\\Desktop\\Studie\\Graduation\\01.
↪THESIS\\Scripts\\wd__directory\\wd__results\\algorithm_results.csv"

# ---------------------------------------------------------------------------
# CSV helper
# ---------------------------------------------------------------------------
def ensure_csv(csv_file: str):
    """
    Ensure the CSV file exists and has a header row.

    Args:
        csv_file: Full path to the CSV file.
    """
    if not os.path.exists(csv_file):
        with open(csv_file, "w", newline="", encoding="utf-8") as f:
            writer = csv.writer(f)
            # Header columns for waterline results
            writer.writerow([
                "image_name",
                "WL_original",
                "WL_hsv",
                "WL_hue",
                "WL_saturation",
                "best_mode",
                "best_score"
            ])

# ---------------------------------------------------------------------------
# Imaging helpers
# ---------------------------------------------------------------------------
def rotate_and_crop(pil_img: Image.Image, angle: float):
    """
    Rotate the PIL image by 'angle' and crop to a predefined box.

    Args:
        pil_img: Input PIL Image in RGB mode.
        angle: Degrees to rotate (positive=CCW).
```

```python
    Returns:
        rotated_np: Full rotated image as numpy array.
        box: Tuple (left, top, right, bottom) of crop box.
        cropped_np: Cropped numpy array region.
    """
    # Fetch crop parameters from config
    cp = CONFIG["crop_params"]
    left, top, right, bottom = cp["left"], cp["top"], cp["right"], cp["bottom"]

    # Rotate and convert to numpy
    rotated = pil_img.rotate(angle, resample=Image.BICUBIC, expand=True)
    r_np = np.array(rotated)
    h, w = r_np.shape[:2]

    # Clamp box within image bounds
    left = max(0, left)
    top = max(0, top)
    right = min(w, right)
    bottom = min(h, bottom)

    # Crop region of interest
    if r_np.ndim == 2:
        cropped_np = r_np[top:bottom, left:right]
    else:
        cropped_np = r_np[top:bottom, left:right, :]

    return r_np, (left, top, right, bottom), cropped_np


def draw_cropbox_on_rotated(rotated_np: np.ndarray, box, out_path: str):
    """
    Create and save an overlay image showing the crop box on the rotated frame.

    Args:
        rotated_np: Numpy array of the rotated image.
        box: (left, top, right, bottom) coordinates.
        out_path: File path to save the visualization PNG.
    """
    left, top, right, bottom = box
    fig, ax = plt.subplots(figsize=(8, 6))

    # Display RGB or grayscale accordingly
    if rotated_np.ndim == 3:
        ax.imshow(rotated_np)
    else:
        ax.imshow(rotated_np, cmap="gray")
```

```python
    # Draw yellow rectangle for crop boundary
    rect = Rectangle((left, top), right-left, bottom-top,
                     edgecolor="yellow", facecolor="none", linewidth=2)
    ax.add_patch(rect)
    ax.set_title("Rotated Image with Crop Box")
    ax.set_xlabel("Column (px)")
    ax.set_ylabel("Row (px)")

    os.makedirs(os.path.dirname(out_path), exist_ok=True)
    plt.savefig(out_path, bbox_inches="tight")
    plt.close(fig)

# ---------------------------------------------------------------------------
# Channel projections & difference metrics
# ---------------------------------------------------------------------------
def get_detection_array(rgb_np: np.ndarray, mode: str):
    """
    Convert image to a single-channel array for difference computation.

    Args:
        rgb_np: Cropped RGB numpy array (H x W x 3).
        mode: One of "original", "hsv", "hue", or "saturation".

    Returns:
        2D array with values in [0,1] for the chosen mode.
    """
    float_np = rgb_np / 255.0
    hsv_img  = rgb_to_hsv(float_np)

    # Select channel based on mode
    if mode == "original":
        return np.mean(rgb_np, axis=2) / 255.0
    if mode == "hsv":
        return hsv_img[:, :, 2]
    if mode == "hue":
        return hsv_img[:, :, 0]
    if mode == "saturation":
        return hsv_img[:, :, 1]
    # Default fallback
    return np.mean(rgb_np, axis=2) / 255.0


def hue_circular_diff(a, b):
    """Compute circular difference for hue values in [0,1]."""
    d = abs(a - b)
    return min(d, 1.0 - d)
```

```python
def mean_diffs(img_np, *, box_h=10, mode="original"):
    """
    Compute vertical mean-difference metric between two adjacent boxes.

    Args:
        img_np: 2D image array in [0,1].
        box_h: Height of comparison boxes in pixels.
        mode: "hue" applies circular diff, else absolute diff.

    Returns:
        List of difference values per possible vertical shift.
    """
    h, _ = img_np.shape
    diffs = []
    for y in range(h - 2*box_h):
        top_box = img_np[y:y+box_h, :]
        bot_box = img_np[y+box_h:y+2*box_h, :]
        m1 = np.mean(top_box)
        m2 = np.mean(bot_box)
        if mode == "hue":
            diffs.append(hue_circular_diff(m1, m2))
        else:
            diffs.append(abs(m1 - m2))
    return diffs


def ks_diffs(img_np, *, box_h=10, mode="original"):
    """
    Compute KS-statistic between two adjacent boxes per vertical shift.

    Args:
        img_np: 2D array of values (e.g., hue sin transform for mode="hue").
        box_h: Height of each box.
        mode: If "hue", transforms via sin for circular data.

    Returns:
        List of KS statistic values per shift.
    """
    h, _ = img_np.shape
    stats = []
    for y in range(h - 2*box_h):
        b1 = img_np[y:y+box_h, :].ravel()
        b2 = img_np[y+box_h:y+2*box_h, :].ravel()
        if mode == "hue":
            # Convert hue to sine wave for circular comparison
            b1 = np.sin(b1 * 2*np.pi)
```

```
            b2 = np.sin(b2 * 2*np.pi)
        stats.append(ks_2samp(b1, b2, mode="asymp")[0])
    return stats

# ---------------------------------------------------------------------------
# Smoothing & peak finding for waterline
# ---------------------------------------------------------------------------
def smoothed_prob(diffs, *, sigma=5):
    """
    Apply Gaussian smoothing and normalize to a probability curve.

    Args:
        diffs: Sequence of difference metric values.
        sigma: Standard deviation for Gaussian kernel.

    Returns:
        Normalized smoothed curve (sums to 1 unless empty).
    """
    if not diffs:
        return []
    s = gaussian_filter1d(np.asarray(diffs), sigma=sigma)
    return s/s.sum() if s.sum() != 0 else np.zeros_like(s)


def primary_peak(p, *, edge=1, min_dist=10):
    """
    Identify the most prominent peak in a 1D probability curve.

    Args:
        p: 1D array of smoothed probabilities.
        edge: Number of elements to zero-out at borders.
        min_dist: Minimum separation between peaks.

    Returns:
        Index of the highest peak, or None if none found.
    """
    p2 = p.copy()
    p2[:edge] = 0
    p2[-edge:] = 0
    peaks, _ = find_peaks(p2, distance=min_dist)
    if not peaks.size:
        return None
    # Pick the peak with maximum probability
    return peaks[np.argmax(p2[peaks])]

# ---------------------------------------------------------------------------
# Visualization helpers
```

```python
# ----------------------------------------------------------------------------
def top_visual(cropped_rgb: np.ndarray, mode: str):
    """
    Prepare top-panel image for each mode in the 2×4 grid.

    Args:
        cropped_rgb: Cropped RGB array (H x W x 3).
        mode: Channel to visualize.

    Returns:
        tv: Image array suitable for imshow.
        cmap: Colormap name (None for RGB).
    """
    if cropped_rgb.ndim != 3 or cropped_rgb.shape[2] != 3:
        return cropped_rgb, "gray"
    float_np = cropped_rgb/255.0
    hsv = rgb_to_hsv(float_np)
    if mode == "original":
        return cropped_rgb, None
    if mode == "hsv":
        return hsv[:,:,2], "gray"
    if mode == "hue":
        return np.sin(hsv[:,:,0]*2*np.pi), "Greys_r"
    if mode == "saturation":
        return hsv[:,:,1], "Greys"
    return cropped_rgb, None

# ----------------------------------------------------------------------------
# 2×4 overview plotting with waterline & validation
# ----------------------------------------------------------------------------
def plot_2x4(
    results, out_path, box_h, top_offset, validation_global=None
):
    """
    Generate and save a 2×4 grid of channel visualizations and metrics.

    Args:
        results: Dict mapping mode -> dict with keys␣
 ↪{tv,cmap,wl,diffs,probs,mids}.
        out_path: File path for saving the overview PNG.
        box_h: Height of comparison boxes in px.
        top_offset: Y-offset of crop top in full image.
        validation_global: Optional ground-truth waterline Y in full image.
    """
    modes = ["original","hsv","hue","saturation"]
    fig, axs = plt.subplots(2,4,figsize=(16,8))
```

```python
    # Compute validation position in cropped coordinates
    if validation_global is not None:
        validation_local = validation_global - top_offset

    for col, m in enumerate(modes):
        r     = results[m]
        tv    = r["tv"]
        cmap  = r["cmap"]
        wl_c  = r["wl"]         # center-of-box Y
        diffs = r["diffs"]
        probs = r["probs"]

        # Build boundary-aligned X axis: midpoints shifted by half box
        mids              = np.arange(box_h+box_h//2, box_h+box_h//2+len(diffs))
        boundary_coords = mids - (box_h//2)
        peak_idx         = primary_peak(probs, edge=1,
                                                 ␣
↪min_dist=CONFIG["processing_params"]["min_distance"])

        # Top image panel
        ax_t = axs[0,col]
        ax_t.imshow(tv if tv.ndim==3 else tv, cmap=None if tv.ndim==3 else cmap)
        ax_t.set_title(m.upper(), fontsize=10)
        ax_t.set_xlabel("Column (px)")
        ax_t.set_ylabel("Row (px)")
        # Detected waterline (red dashed) at boundary
        if wl_c is not None:
            ax_t.axhline(wl_c-box_h//2, linestyle="--",
                         color="red", linewidth=2,
                         label="Detected Waterline")
        # Validation line (ground truth)
        if validation_global is not None:
            ax_t.axhline(validation_local,
                         linestyle=":", color="green",
                         linewidth=2, label="Validation Waterline")

        # Bottom metrics panel
        ax_b = axs[1,col]
        ax_b.plot(boundary_coords, diffs, linewidth=1, label="Diff Metric")
        ax2  = ax_b.twinx()
        ax2.plot(boundary_coords, probs,  linewidth=1, label="Smoothed Prob")
        # Mark detected peak
        if peak_idx is not None:
            xpk = boundary_coords[peak_idx]
            ax_b.axvline(xpk, linestyle="--", color="red", linewidth=2)
            ax2.axvline(xpk, linestyle="--", color="red", linewidth=2)
        # Mark validation peak
```

```
        if validation_global is not None:
            idx = np.argmin(np.abs(boundary_coords-validation_local))
            xgt = boundary_coords[idx]
            ax_b.axvline(xgt, linestyle=":", color="green", linewidth=2)
            ax2.axvline(xgt, linestyle=":", color="green", linewidth=2)

        ax_b.set_title(f"{m.upper()} Channel: Diff vs Smoothed Prob",␣
↪fontsize=9)
        ax_b.set_xlabel("Pixel Row (px)")
        ax_b.set_ylabel("Difference Metric (unitless)")
        ax2.set_ylabel("Smoothed Probability (unitless)")
        # Legends
        ax_b.legend(loc="upper left", fontsize=7)
        ax2.legend(loc="upper left", bbox_to_anchor=(0,0.9), fontsize=7)

    plt.tight_layout()
    os.makedirs(os.path.dirname(out_path), exist_ok=True)
    plt.savefig(out_path)
    plt.close(fig)
    logger.info(f"[FIG] saved -> {out_path}")

# ---------------------------------------------------------------------------
# Main processing routine
# ---------------------------------------------------------------------------
def process_image(
    image_path: str,
    output_folder: str,
    angle: float,
    box_height: int,
    min_distance: int,
    sigma: int,
    *,
    system: str = "raspberry_pi",
):
    """
    Full pipeline to detect waterline in a single image.

    Steps:
      1. Prepare CSV logging
      2. Load and rotate image; crop region
      3. Compute diffs & probs for each mode
      4. Identify best waterline and save visuals
      5. Append results to CSV and copy overview to USB

    Args:
        image_path: Path to input JPEG image.
        output_folder: Directory to save visuals.
```

```
        angle: Rotation angle to deskew.
        box_height: Height of comparison window.
        min_distance: Peak separation for detection.
        sigma: Smoothing parameter.
        system: "raspberry_pi" or "pc" (chooses CSV path).

    Returns:
        Tuple (overview_png_path, {"primary_index": row_px}) or None on error.
    """
    try:
        # Ensure CSV file with header exists
        csv_file = CSV_FILE_PI if system=="raspberry_pi" else CSV_FILE_PC
        ensure_csv(csv_file)

        # Determine diff method from config ("ks" or "mean")
        diff_method = CONFIG["processing_params"].get("diff_method", "ks")
        pil_img = Image.open(image_path).convert("RGB")

        # Rotate and crop image, get bounding box info
        rot_np, box, cropped = rotate_and_crop(pil_img, angle)
        left, top, _, _ = box

        # Prepare filenames and suffix based on method
        base, ext = os.path.splitext(os.path.basename(image_path))
        suffix = "_KS" if diff_method=="ks" else ""
        base_out = f"{base}{suffix}"

        # Save cropbox overlay visualization
        cropbox_png = os.path.join(output_folder, f"{base_out}_cropbox.png")
        draw_cropbox_on_rotated(rot_np, box, cropbox_png)

        # Process each channel/mode
        modes = ["original","hsv","hue","saturation"]
        results = {}
        for m in modes:
            # Get single-channel array
            arr = get_detection_array(cropped, m) if cropped.ndim==3 else␣
↪cropped/255.0
            # Compute diffs then smoothed probability
            metric_fn = ks_diffs if diff_method=="ks" else mean_diffs
            diffs = metric_fn(arr, box_h=box_height, mode=m)
            probs = smoothed_prob(diffs, sigma=sigma)
            # Build center-of-box mids array
            mids = np.arange(box_height+box_height//2,
                             box_height+box_height//2+len(diffs))
            peak = primary_peak(probs, edge=1, min_dist=min_distance)
            wl_c = None if peak is None else mids[peak]
```

```
        score = max(diffs) if diffs else 0.0
        tv, cmap = top_visual(cropped, m)
        results[m] = {"wl":wl_c, "diffs":diffs,
                      "probs":probs, "mids":mids,
                      "score":score, "tv":tv, "cmap":cmap}

    # Ground-truth line (for validation overlay)
    VALIDATION_GLOBAL = CONFIG.get("validation_global", None)
    # Generate overview 2×4 plot
    overview_png = os.path.join(output_folder, f"{base_out}_4modes.png")
    plot_2x4(results, overview_png, box_height, top,
            validation_global=VALIDATION_GLOBAL)

    # Determine best mode and compute boundary-based WL in full image coords
    best_mode = max(modes, key=lambda m: results[m]["score"])
    wl_best_c = results[best_mode]["wl"]
    wl_best_r = None if wl_best_c is None else wl_best_c - box_height//2 + top

    # Append results to CSV
    img_name_csv = f"{base_out}{ext}"
    wl_cols = [
        (results[m]["wl"]-box_height//2+top) if results[m]["wl"] is not None else ""
        for m in modes
    ]
    with open(csv_file, "a", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            img_name_csv,
            *(f"{v:.2f}" for v in wl_cols if v!=""),
            best_mode,
            f"{results[best_mode]["score"]:.2f}"
        ])

    # Copy overview to USB if mounted
    if wl_best_r is not None:
        usb = "/media/bjorn/C0D9-1D92"
        try:
            os.makedirs(usb, exist_ok=True)
            shutil.copy2(overview_png, os.path.join(usb, os.path.basename(overview_png)))
            logger.info(f"Copied overview to USB: {usb}")
        except Exception as e:
            logger.warning(f"USB copy failed: {e}")
```

```python
        return overview_png, {"primary_index": None if wl_best_r is None else␣
↪int(wl_best_r)}

    except Exception as e:
        logger.error(f"[ERROR] processing {image_path}: {e}", exc_info=True)
        return None

# End of module
```