**Remote Sensing – Plot-Level ET Estimation**

## **1. Abstract**

## Objectives -

* To estimate daily evapotranspiration (ET) for 30 plots using WaPOR data.
* Deal with spatial resolution and temporal interpolation challenges.
* Compare satellite-based ET with ground-truth water usage from meters.

### **2.1. WaPOR Data**

* Source: FAO WaPOR Level-1 (Dekadal ET, 300m resolution).
* Units: mm/day (dekadal sums provided).

### **2.2. Plot Information**

* 30 plots across different farm management types:
  + TPR (Transplanted Rice)
  + DSR (Direct Seeded Rice)
  + AWD (Alternate Wetting and Drying)
* Plots are smaller than pixel resolution (100–200 m in size or 2-5 acres).

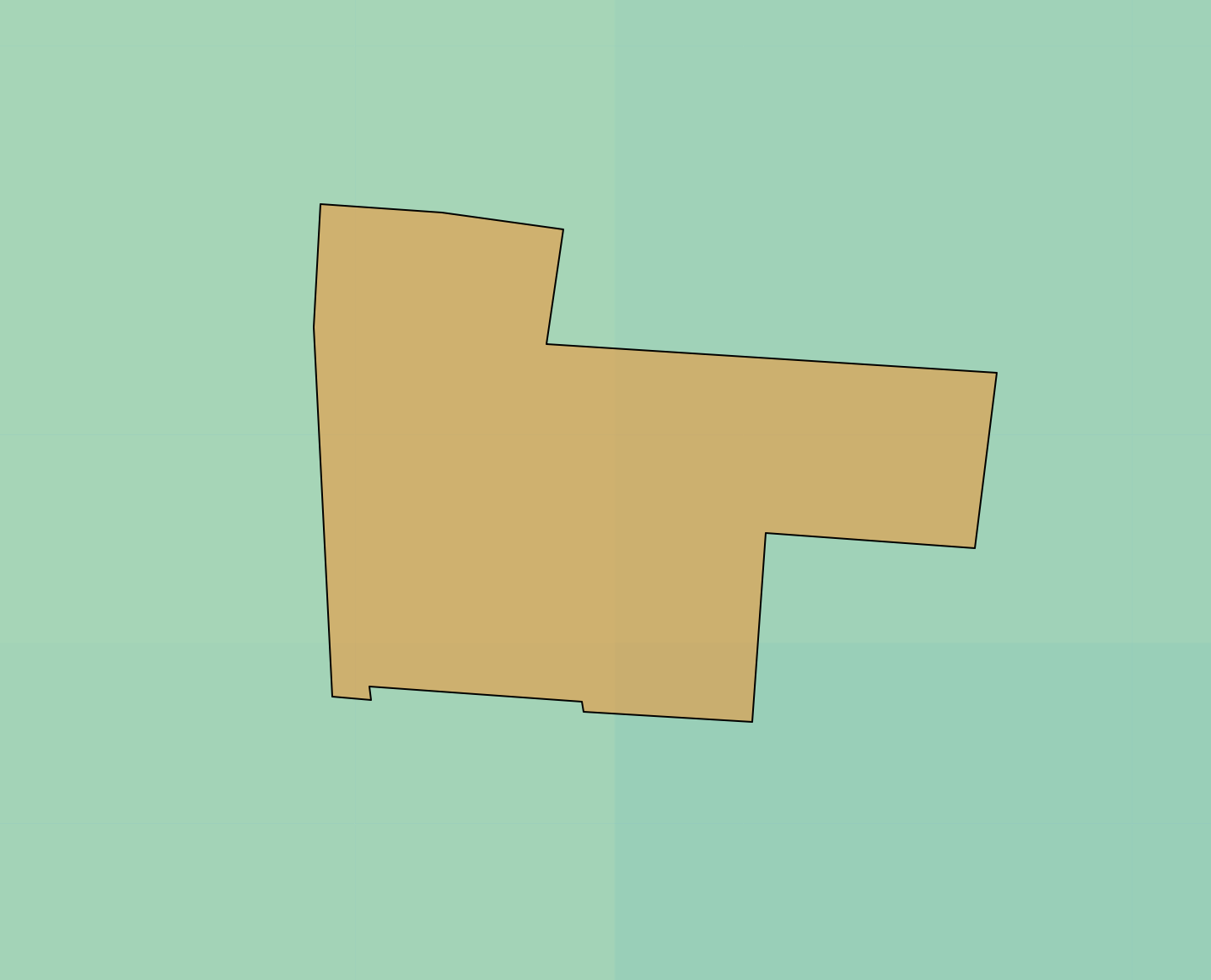
Link to Sample plots - <https://docs.google.com/spreadsheets/d/1UrMY4plAIWYo_-UX7tvCAZfd4BBbuUAYrSaKru6elbY/edit?usp=sharing>

## **3. Methodology**

### **3.1. Pixel-Plot Overlap Resolution**

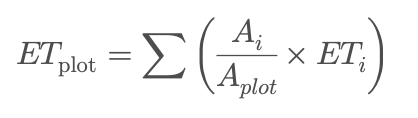
#### **Problem:**

WaPOR pixels (300m) are coarser than plot boundaries, causing multiple pixel overlaps.

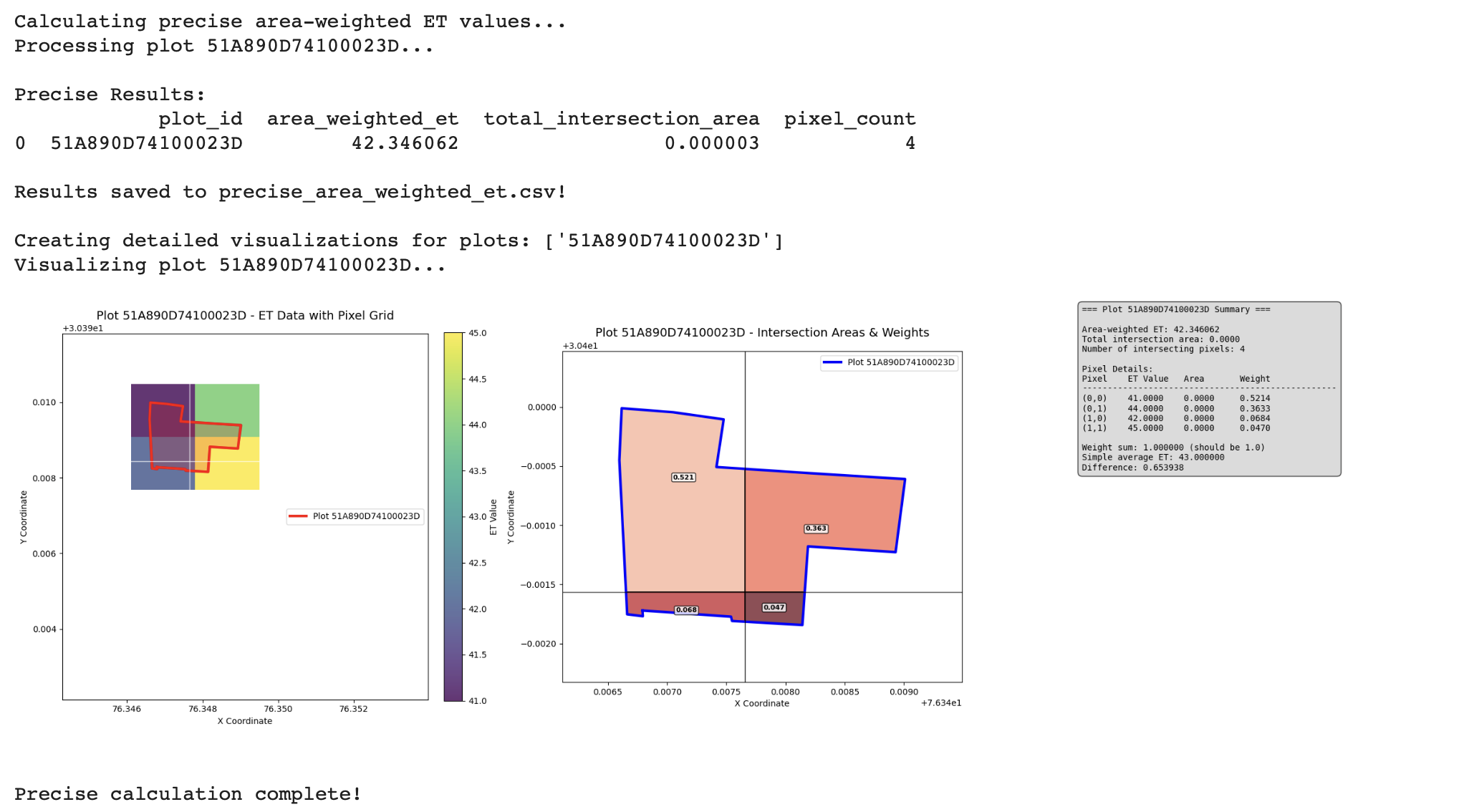


(plot id = LA\_PA\_PA\_HARPA\_PLT\_5052)

#### **Solution: Area-weighted averaging**

* Overlay plot polygons on WaPOR raster.
* Calculate intersection area of each pixel with plot using **Shoelace Algorithm** (Green’s Theorem) or GIS tools.
* Multiply pixel ET by area fraction and sum:  
  

Output -



#### **Validation:**

* **Area-weighted averaging** is widely used in downscaling satellite data to field level.
* Similar approach used in:
  + FAO’s WaPOR methodology documentation:

<https://wapor.apps.fao.org/home/WAPOR_Methodology>

* + NASA Land Data Assimilation Systems (LDAS) - Downscaling

*“For point or plot-scale validation, the average value of the intersecting pixels weighted by the fraction of coverage should be used.”*

<https://ldas.gsfc.nasa.gov/>

* + QGIS Documentation – Zonal Statistics

<https://docs.qgis.org/>

### **3.2. Temporal Interpolation**

#### Problem:

ET data is dekadal (e.g., 12 mm every 10 days) — coarse for daily water analysis.

#### Solution: Cubic Spline Interpolation

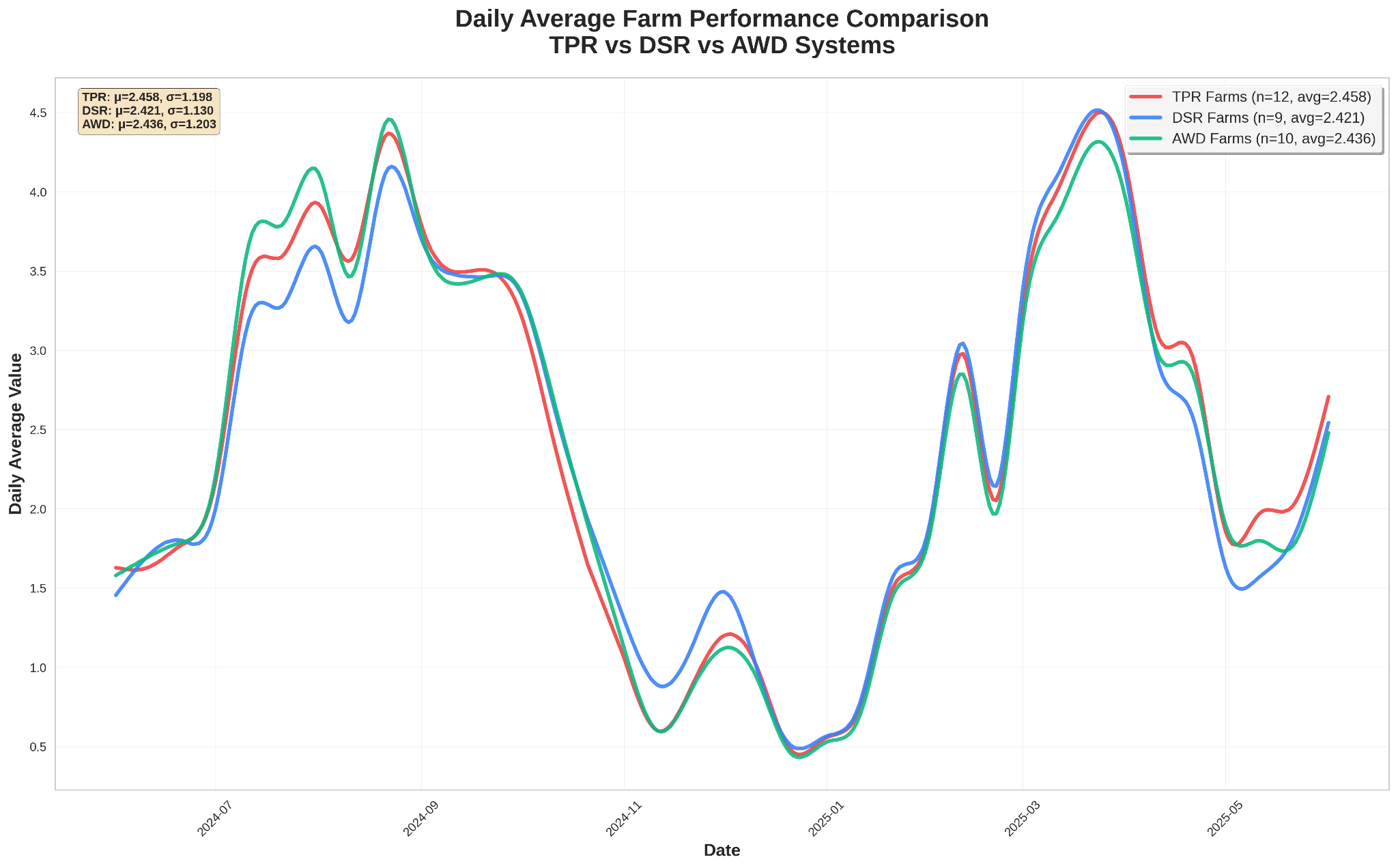
Instead of assuming constant ET/day = total/10 (which is unrealistic), cubic spline interpolation was applied to smooth values across time.

Cubic spline interpolation is a method to construct a **smooth curve** that passes through a series of known data points. Unlike linear or polynomial interpolation, a **cubic spline** stitches together **piecewise cubic polynomials** between each pair of points, ensuring smooth transitions at the joints.

**Spline interpolation creates daily or finer-resolution estimates** from coarse data while maintaining realistic trends.This method better captures temporal variability, aligns with weather-induced ET shifts.

Validation:

* Cubic splines are commonly used for temporal interpolation in environmental datasets.
* Used in ET studies like:
  + [Zhang et al., 2017 – Journal of Hydrology](https://doi.org/10.1016/j.jhydrol.2017.06.005)
  + NASA EarthData and MODIS preprocessing pipelines.



(all values in mm/day)

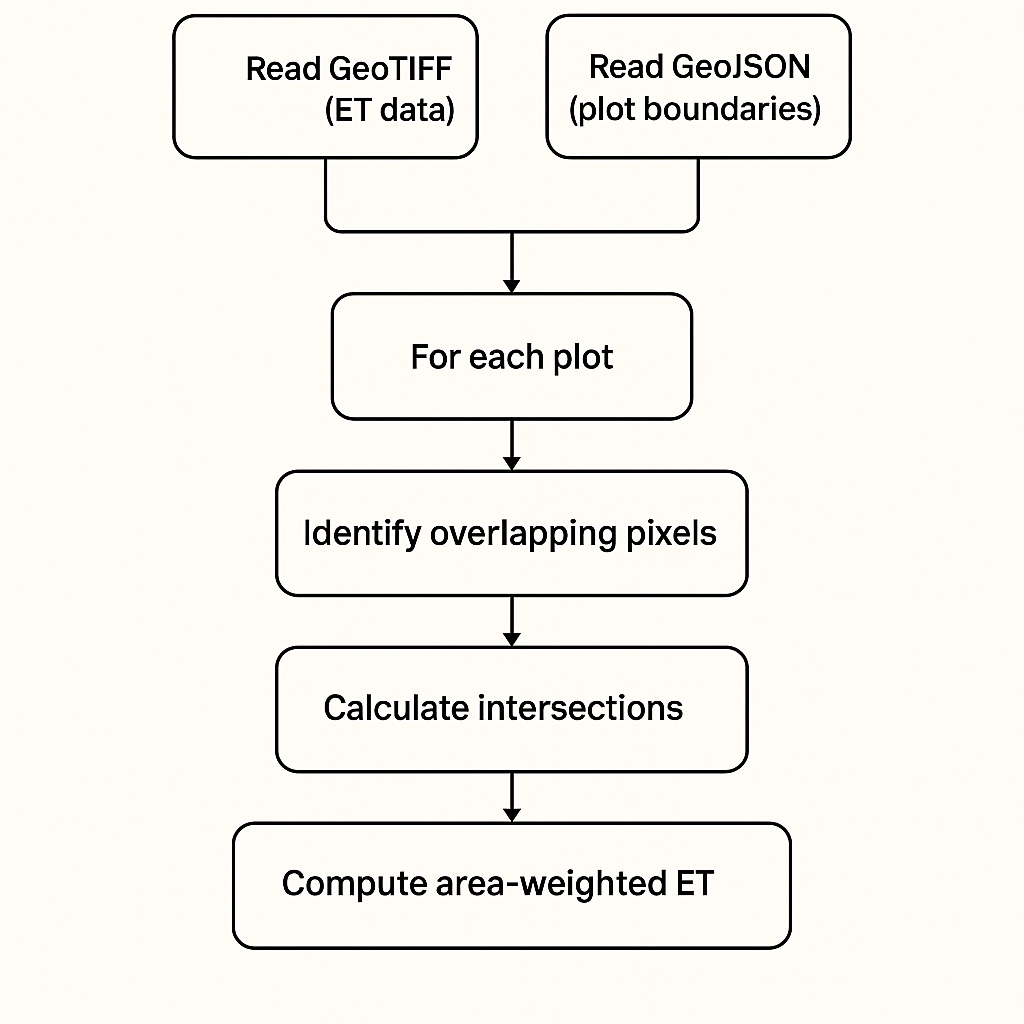
## **4. Visualizations and Analysis**

* Daily ET curves per plot.
* Boxplots comparing ET distributions across TPR, DSR, and AWD.
* Whisker plots to show variability across seasons or plots.
* Comparative bar charts showing peak vs. low ET periods.

### **Tools Used:**

* Python (Matplotlib, Seaborn, Scipy for interpolation)

**SUM UP**

****

Flowchart of logic

Link to the final data (1 JUNE 2024 - 1 JUNE 2025) : <https://docs.google.com/spreadsheets/d/1BIZ1y6_kQP5UEn6RG4NCZykfGErjCuwx8VpyyTQAMPo/edit?usp=sharing>

## **5. Code Snippets**

### **5.1 Area weighted average**

import rasterio

import geopandas as gpd

import numpy as np

from rasterio.features import rasterize

from rasterio.transform import from\_bounds

from shapely.geometry import box, Polygon

from shapely.ops import unary\_union

import pandas as pd

from typing import Dict, List, Tuple, Optional

import warnings

import matplotlib.pyplot as plt

from matplotlib.patches import Polygon as MPLPolygon

warnings.filterwarnings('ignore')

def shoelace\_area(coordinates: List[Tuple[float, float]]) -> float:

"""

Calculate polygon area using the shoelace formula.

Parameters:

-----------

coordinates : List[Tuple[float, float]]

List of (x, y) coordinate tuples forming the polygon

Returns:

--------

float

Area of the polygon

"""

if len(coordinates) < 3:

return 0.0

# Ensure polygon is closed

if coordinates[0] != coordinates[-1]:

coordinates = coordinates + [coordinates[0]]

n = len(coordinates) - 1 # Exclude the duplicate closing point

area = 0.0

for i in range(n):

j = (i + 1) % n

area += coordinates[i][0] \* coordinates[j][1]

area -= coordinates[j][0] \* coordinates[i][1]

return abs(area) / 2.0

def get\_pixel\_coordinates(pixel\_row: int, pixel\_col: int, transform: rasterio.Affine) -> List[Tuple[float, float]]:

"""

Get the corner coordinates of a pixel.

Parameters:

-----------

pixel\_row : int

Row index of the pixel

pixel\_col : int

Column index of the pixel

transform : rasterio.Affine

Raster transform

Returns:

--------

List[Tuple[float, float]]

List of corner coordinates [(x, y), ...]

"""

# Get pixel bounds in geographic coordinates

left = transform \* (pixel\_col, pixel\_row)

right = transform \* (pixel\_col + 1, pixel\_row)

bottom\_left = transform \* (pixel\_col, pixel\_row + 1)

bottom\_right = transform \* (pixel\_col + 1, pixel\_row + 1)

# Return corners in order: top-left, top-right, bottom-right, bottom-left

return [

left, # top-left

right, # top-right

bottom\_right, # bottom-right

bottom\_left # bottom-left

]

def get\_intersection\_coordinates(plot\_geometry, pixel\_coords: List[Tuple[float, float]]) -> Optional[List[Tuple[float, float]]]:

"""

Get the coordinates of intersection between plot geometry and pixel.

Parameters:

-----------

plot\_geometry : shapely.geometry

Plot geometry

pixel\_coords : List[Tuple[float, float]]

Pixel corner coordinates

Returns:

--------

Optional[List[Tuple[float, float]]]

Intersection coordinates or None if no intersection

"""

try:

# Create pixel polygon

pixel\_polygon = Polygon(pixel\_coords)

# Calculate intersection

intersection = plot\_geometry.intersection(pixel\_polygon)

if intersection.is\_empty:

return None

# Handle different geometry types

if hasattr(intersection, 'exterior'):

# Single polygon

return list(intersection.exterior.coords[:-1]) # Exclude duplicate last point

elif hasattr(intersection, 'geoms'):

# MultiPolygon or GeometryCollection

all\_coords = []

for geom in intersection.geoms:

if hasattr(geom, 'exterior'):

all\_coords.extend(list(geom.exterior.coords[:-1]))

return all\_coords if all\_coords else None

else:

return None

except Exception as e:

print(f"Error calculating intersection: {e}")

return None

def calculate\_precise\_area\_weighted\_et(geotiff\_path: str, geojson\_path: str,

plot\_id\_column: str = 'id') -> pd.DataFrame:

"""

Calculate area-weighted ET values using precise coordinate-based area calculation.

Parameters:

-----------

geotiff\_path : str

Path to the GeoTIFF file containing ET values

geojson\_path : str

Path to the GeoJSON file containing plot boundaries

plot\_id\_column : str

Column name in GeoJSON that contains plot identifiers

Returns:

--------

pd.DataFrame

DataFrame with plot IDs and their area-weighted ET values

"""

# Read the GeoJSON file

plots\_gdf = gpd.read\_file(geojson\_path)

# Read the GeoTIFF file

with rasterio.open(geotiff\_path) as src:

et\_data = src.read(1) # Read first band

transform = src.transform

crs = src.crs

nodata = src.nodata

# Ensure both datasets have the same CRS

if plots\_gdf.crs != crs:

plots\_gdf = plots\_gdf.to\_crs(crs)

results = []

for idx, plot in plots\_gdf.iterrows():

plot\_id = plot[plot\_id\_column] if plot\_id\_column in plot else idx

geometry = plot.geometry

print(f"Processing plot {plot\_id}...")

# Get the bounding box of the plot

minx, miny, maxx, maxy = geometry.bounds

# Convert bounds to pixel coordinates

ul\_col, ul\_row = ~transform \* (minx, maxy) # Upper left

lr\_col, lr\_row = ~transform \* (maxx, miny) # Lower right

# Round to get pixel indices

ul\_col, ul\_row = int(np.floor(ul\_col)), int(np.floor(ul\_row))

lr\_col, lr\_row = int(np.ceil(lr\_col)), int(np.ceil(lr\_row))

# Ensure indices are within raster bounds

ul\_col = max(0, ul\_col)

ul\_row = max(0, ul\_row)

lr\_col = min(et\_data.shape[1], lr\_col)

lr\_row = min(et\_data.shape[0], lr\_row)

if ul\_col >= lr\_col or ul\_row >= lr\_row:

print(f"Warning: Plot {plot\_id} is outside raster bounds")

results.append({

'plot\_id': plot\_id,

'area\_weighted\_et': np.nan,

'total\_intersection\_area': 0,

'pixel\_count': 0,

'intersection\_details': []

})

continue

# Store intersection details for each pixel

intersection\_details = []

total\_intersection\_area = 0

# Process each pixel in the bounding box

for row in range(ul\_row, lr\_row):

for col in range(ul\_col, lr\_col):

# Get pixel value

pixel\_value = et\_data[row, col]

# Skip nodata pixels

if nodata is not None and pixel\_value == nodata:

continue

if np.isnan(pixel\_value):

continue

# Get pixel corner coordinates

pixel\_coords = get\_pixel\_coordinates(row, col, transform)

# Get intersection coordinates

intersection\_coords = get\_intersection\_coordinates(geometry, pixel\_coords)

if intersection\_coords and len(intersection\_coords) >= 3:

# Calculate intersection area using shoelace formula

intersection\_area = shoelace\_area(intersection\_coords)

if intersection\_area > 0:

intersection\_details.append({

'pixel\_row': row,

'pixel\_col': col,

'pixel\_value': pixel\_value,

'intersection\_area': intersection\_area,

'intersection\_coords': intersection\_coords,

'pixel\_coords': pixel\_coords

})

total\_intersection\_area += intersection\_area

# Calculate area-weighted ET

if total\_intersection\_area > 0 and intersection\_details:

area\_weighted\_sum = 0

# Calculate weights and weighted sum

for detail in intersection\_details:

weight = detail['intersection\_area'] / total\_intersection\_area

detail['weight'] = weight

area\_weighted\_sum += detail['pixel\_value'] \* weight

area\_weighted\_et = area\_weighted\_sum

else:

area\_weighted\_et = np.nan

results.append({

'plot\_id': plot\_id,

'area\_weighted\_et': area\_weighted\_et,

'total\_intersection\_area': total\_intersection\_area,

'pixel\_count': len(intersection\_details),

'intersection\_details': intersection\_details

})

# Create DataFrame with main results

main\_results = pd.DataFrame([{

'plot\_id': r['plot\_id'],

'area\_weighted\_et': r['area\_weighted\_et'],

'total\_intersection\_area': r['total\_intersection\_area'],

'pixel\_count': r['pixel\_count']

} for r in results])

# Store detailed results for visualization

main\_results.\_detailed\_results = results

return main\_results

def visualize\_precise\_plot\_coverage(geotiff\_path: str, geojson\_path: str, plot\_id: str,

results\_df: pd.DataFrame, plot\_id\_column: str = 'id'):

"""

Visualize precise plot coverage with coordinate-based intersections.

"""

# Get detailed results

detailed\_results = getattr(results\_df, '\_detailed\_results', None)

if detailed\_results is None:

print("No detailed results available. Please run calculate\_precise\_area\_weighted\_et first.")

return

# Find the specific plot results

plot\_result = None

for result in detailed\_results:

if str(result['plot\_id']) == str(plot\_id):

plot\_result = result

break

if plot\_result is None:

print(f"Plot {plot\_id} not found in results.")

return

# Read data

plots\_gdf = gpd.read\_file(geojson\_path)

plot = plots\_gdf[plots\_gdf[plot\_id\_column] == plot\_id].iloc[0]

with rasterio.open(geotiff\_path) as src:

et\_data = src.read(1)

transform = src.transform

crs = src.crs

nodata = src.nodata

# Ensure same CRS

if plots\_gdf.crs != crs:

plots\_gdf = plots\_gdf.to\_crs(crs)

plot = plots\_gdf[plots\_gdf[plot\_id\_column] == plot\_id].iloc[0]

# Create visualization

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 8))

# Get plot bounds with buffer

minx, miny, maxx, maxy = plot.geometry.bounds

buffer = max((maxx - minx), (maxy - miny)) \* 0.2

plot\_minx, plot\_miny = minx - buffer, miny - buffer

plot\_maxx, plot\_maxy = maxx + buffer, maxy + buffer

# Convert to pixel coordinates for cropping

ul\_col, ul\_row = ~transform \* (plot\_minx, plot\_maxy)

lr\_col, lr\_row = ~transform \* (plot\_maxx, plot\_miny)

ul\_col, ul\_row = int(np.floor(ul\_col)), int(np.floor(ul\_row))

lr\_col, lr\_row = int(np.ceil(lr\_col)), int(np.ceil(lr\_row))

ul\_col = max(0, ul\_col)

ul\_row = max(0, ul\_row)

lr\_col = min(et\_data.shape[1], lr\_col)

lr\_row = min(et\_data.shape[0], lr\_row)

# Create extent

extent = [plot\_minx, plot\_maxx, plot\_miny, plot\_maxy]

# Crop raster

subset = et\_data[ul\_row:lr\_row, ul\_col:lr\_col]

subset\_masked = np.copy(subset).astype(float)

if nodata is not None:

subset\_masked[subset == nodata] = np.nan

# Plot 1: ET data with plot overlay

im1 = ax1.imshow(subset\_masked, extent=extent, origin='upper', alpha=0.8, cmap='viridis')

# Plot plot boundary

if hasattr(plot.geometry, 'exterior'):

x, y = plot.geometry.exterior.xy

ax1.plot(x, y, color='red', linewidth=3, label=f'Plot {plot\_id}')

ax1.fill(x, y, color='red', alpha=0.2)

# Add pixel grid

for row in range(ul\_row, lr\_row + 1):

for col in range(ul\_col, lr\_col + 1):

pixel\_coords = get\_pixel\_coordinates(row, col, transform)

if pixel\_coords:

pixel\_x = [coord[0] for coord in pixel\_coords] + [pixel\_coords[0][0]]

pixel\_y = [coord[1] for coord in pixel\_coords] + [pixel\_coords[0][1]]

ax1.plot(pixel\_x, pixel\_y, color='white', alpha=0.5, linewidth=1)

ax1.set\_title(f'Plot {plot\_id} - ET Data with Pixel Grid', fontsize=14)

ax1.set\_xlabel('X Coordinate')

ax1.set\_ylabel('Y Coordinate')

ax1.legend()

plt.colorbar(im1, ax=ax1, label='ET Value', fraction=0.046, pad=0.04)

# Plot 2: Intersection areas

ax2.set\_xlim(plot\_minx, plot\_maxx)

ax2.set\_ylim(plot\_miny, plot\_maxy)

# Plot plot boundary

if hasattr(plot.geometry, 'exterior'):

x, y = plot.geometry.exterior.xy

ax2.plot(x, y, color='blue', linewidth=3, label=f'Plot {plot\_id}')

# Plot intersection areas with different colors based on weights

colors = plt.cm.Reds(np.linspace(0.3, 1.0, len(plot\_result['intersection\_details'])))

for i, detail in enumerate(plot\_result['intersection\_details']):

# Plot pixel boundary

pixel\_coords = detail['pixel\_coords']

pixel\_x = [coord[0] for coord in pixel\_coords] + [pixel\_coords[0][0]]

pixel\_y = [coord[1] for coord in pixel\_coords] + [pixel\_coords[0][1]]

ax2.plot(pixel\_x, pixel\_y, color='black', linewidth=1, alpha=0.5)

# Plot intersection area

intersection\_coords = detail['intersection\_coords']

if len(intersection\_coords) >= 3:

intersection\_x = [coord[0] for coord in intersection\_coords]

intersection\_y = [coord[1] for coord in intersection\_coords]

# Create matplotlib polygon

polygon = MPLPolygon(list(zip(intersection\_x, intersection\_y)),

facecolor=colors[i], alpha=0.7, edgecolor='black', linewidth=1)

ax2.add\_patch(polygon)

# Add weight text

centroid\_x = np.mean(intersection\_x)

centroid\_y = np.mean(intersection\_y)

ax2.text(centroid\_x, centroid\_y, f'{detail["weight"]:.3f}',

ha='center', va='center', fontsize=8, fontweight='bold',

bbox=dict(boxstyle='round,pad=0.2', facecolor='white', alpha=0.8))

ax2.set\_title(f'Plot {plot\_id} - Intersection Areas & Weights', fontsize=14)

ax2.set\_xlabel('X Coordinate')

ax2.set\_ylabel('Y Coordinate')

ax2.legend()

ax2.set\_aspect('equal')

# Plot 3: Summary statistics

ax3.axis('off')

# Prepare summary data

summary\_text = f"=== Plot {plot\_id} Summary ===\n\n"

summary\_text += f"Area-weighted ET: {plot\_result['area\_weighted\_et']:.6f}\n"

summary\_text += f"Total intersection area: {plot\_result['total\_intersection\_area']:.4f}\n"

summary\_text += f"Number of intersecting pixels: {plot\_result['pixel\_count']}\n\n"

summary\_text += "Pixel Details:\n"

summary\_text += f"{'Pixel':<8} {'ET Value':<10} {'Area':<10} {'Weight':<10}\n"

summary\_text += f"{'-'\*50}\n"

for i, detail in enumerate(plot\_result['intersection\_details']):

pixel\_pos = f"({detail['pixel\_row']},{detail['pixel\_col']})"

summary\_text += f"{pixel\_pos:<8} {detail['pixel\_value']:<10.4f} "

summary\_text += f"{detail['intersection\_area']:<10.4f} {detail['weight']:<10.4f}\n"

# Verification

total\_weight = sum(detail['weight'] for detail in plot\_result['intersection\_details'])

summary\_text += f"\nWeight sum: {total\_weight:.6f} (should be 1.0)\n"

# Simple average for comparison

pixel\_values = [detail['pixel\_value'] for detail in plot\_result['intersection\_details']]

if pixel\_values:

simple\_avg = np.mean(pixel\_values)

summary\_text += f"Simple average ET: {simple\_avg:.6f}\n"

summary\_text += f"Difference: {abs(plot\_result['area\_weighted\_et'] - simple\_avg):.6f}"

ax3.text(0.05, 0.95, summary\_text, transform=ax3.transAxes, fontsize=10,

verticalalignment='top', fontfamily='monospace',

bbox=dict(boxstyle='round,pad=0.5', facecolor='lightgray', alpha=0.8))

plt.tight\_layout()

plt.show()

return fig

def compare\_methods(geotiff\_path: str, geojson\_path: str, plot\_id\_column: str = 'id'):

"""

Compare the original and precise methods.

"""

print("Running original method...")

# Note: You'll need to import your original function

# original\_results = calculate\_area\_weighted\_et(geotiff\_path, geojson\_path, plot\_id\_column)

print("Running precise method...")

precise\_results = calculate\_precise\_area\_weighted\_et(geotiff\_path, geojson\_path, plot\_id\_column)

print("\nPrecise method results:")

print(precise\_results.head())

return precise\_results

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# File paths

geotiff\_path = "/content/5052\_01\_04\_25"

geojson\_path = "/content/plot\_5052.json"

print("Calculating precise area-weighted ET values...")

# Calculate using precise coordinate-based method

results = calculate\_precise\_area\_weighted\_et(

geotiff\_path,

geojson\_path,

plot\_id\_column='id'

)

# Display results

print("\nPrecise Results:")

print(results)

# Save results to CSV

results.to\_csv('precise\_area\_weighted\_et.csv', index=False)

print("\nResults saved to precise\_area\_weighted\_et.csv!")

# Visualize specific plots

if len(results) > 0:

plot\_ids = results['plot\_id'].head(2).tolist()

print(f"\nCreating detailed visualizations for plots: {plot\_ids}")

for plot\_id in plot\_ids:

print(f"Visualizing plot {plot\_id}...")

visualize\_precise\_plot\_coverage(geotiff\_path, geojson\_path, str(plot\_id),

results, plot\_id\_column='id')

print("\nPrecise calculation complete!")

### **5.2 Temporal Interpolation (Cubic spline)**

import pandas as pd

import numpy as np

from scipy.interpolate import CubicSpline

import matplotlib.pyplot as plt

import matplotlib.dates as mdates

from datetime import datetime

import warnings

import os

# Suppress warnings for cleaner output

warnings.filterwarnings('ignore')

def test\_file\_access():

"""Test if the Excel file is accessible"""

test\_file = '/content/june-oct.xlsx'

print("="\*60)

print("CHECKING FILE ACCESS...")

print("="\*60)

if os.path.exists(test\_file):

print(f"✓ File found: {test\_file}")

return test\_file

else:

print(f"✗ File not found: {test\_file}")

# Check for other Excel files

try:

files = os.listdir('/content/')

excel\_files = [f for f in files if f.endswith(('.xlsx', '.xls'))]

if excel\_files:

print("\nAvailable Excel files in /content/:")

for f in excel\_files:

print(f" - {f}")

# Use the first Excel file found

return f'/content/{excel\_files[0]}'

else:

print("No Excel files found in /content/")

print("Please upload your Excel file to the Colab environment")

return None

except:

print("Could not access /content/ directory")

return None

def load\_farm\_type\_tables(file\_path):

"""Load the three farm type tables (TPR, DSR, AWD) from Excel file."""

# Try to read from different sheet names

try:

df\_raw = pd.read\_excel(file\_path, sheet\_name='Sheet1', header=None)

except:

try:

df\_raw = pd.read\_excel(file\_path, sheet\_name='Sheet2', header=None)

except:

df\_raw = pd.read\_excel(file\_path, header=None)

# Find table boundaries automatically

table\_starts = []

for idx, row in df\_raw.iterrows():

if any(str(val).lower().strip() == 'date' for val in row.values if pd.notna(val)):

table\_starts.append(idx)

print(f"Found {len(table\_starts)} table headers at rows: {[x+1 for x in table\_starts]}")

# Define table configurations

farm\_type\_names = ['TPR', 'DSR', 'AWD']

farm\_tables = {}

for i, start\_row in enumerate(table\_starts):

farm\_type = farm\_type\_names[i] if i < len(farm\_type\_names) else f'Table\_{i+1}'

# Determine end row

if i + 1 < len(table\_starts):

end\_row = table\_starts[i + 1]

else:

end\_row = len(df\_raw)

for j in range(start\_row + 1, len(df\_raw)):

if df\_raw.iloc[j].isna().all():

has\_more\_data = False

for k in range(j + 1, min(j + 5, len(df\_raw))):

if not df\_raw.iloc[k].isna().all():

has\_more\_data = True

break

if not has\_more\_data:

end\_row = j

break

# Extract headers and data

headers = df\_raw.iloc[start\_row].values

headers = [str(h) if pd.notna(h) else f'col\_{i}' for i, h in enumerate(headers)]

data\_start = start\_row + 1

data\_rows = df\_raw.iloc[data\_start:end\_row].copy()

data\_rows = data\_rows.dropna(how='all')

if not data\_rows.empty:

data\_rows.columns = headers[:len(data\_rows.columns)]

table\_df = process\_farm\_table(data\_rows, farm\_type)

if not table\_df.empty:

farm\_tables[farm\_type] = table\_df

print(f"✓ {farm\_type}: {len(table\_df)} dekadal periods, {len(table\_df.columns)} farms")

return farm\_tables

def process\_farm\_table(table\_data, farm\_type):

"""Process individual farm table."""

table\_data = table\_data.copy()

# Find date column

date\_col = None

for col in table\_data.columns:

if 'date' in str(col).lower():

date\_col = col

break

if date\_col is None:

date\_col = table\_data.columns[0]

# Convert date column

try:

table\_data[date\_col] = pd.to\_datetime(table\_data[date\_col], errors='coerce')

except:

try:

# Handle Excel serial numbers

table\_data[date\_col] = pd.to\_datetime(table\_data[date\_col], origin='1899-12-30', unit='D', errors='coerce')

except:

print(f"Warning: Could not convert dates for {farm\_type}")

return pd.DataFrame()

# Remove rows with invalid dates

table\_data = table\_data.dropna(subset=[date\_col])

if table\_data.empty:

return pd.DataFrame()

# Set date as index

table\_data.set\_index(date\_col, inplace=True)

# Convert other columns to numeric

numeric\_columns = []

for col in table\_data.columns:

if col != date\_col:

try:

if isinstance(table\_data[col], pd.Series):

table\_data[col] = pd.to\_numeric(table\_data[col], errors='coerce')

if table\_data[col].notna().sum() > 0:

numeric\_columns.append(col)

except:

continue

if not numeric\_columns:

return pd.DataFrame()

table\_data = table\_data[numeric\_columns]

table\_data = table\_data.dropna(axis=1, how='all')

return table\_data

def interpolate\_to\_daily(dekadal\_df, farm\_type):

"""Convert dekadal data to daily using cubic spline interpolation."""

if dekadal\_df.empty:

return pd.DataFrame()

dekadal\_df = dekadal\_df.sort\_index()

start\_date = dekadal\_df.index.min()

end\_date = dekadal\_df.index.max()

daily\_dates = pd.date\_range(start=start\_date, end=end\_date, freq='D')

dekadal\_numeric = (dekadal\_df.index - start\_date).days.values

daily\_numeric = (daily\_dates - start\_date).days.values

daily\_df = pd.DataFrame(index=daily\_dates)

print(f" Interpolating {farm\_type}: {len(dekadal\_df)} → {len(daily\_dates)} days")

successful\_farms = []

for col in dekadal\_df.columns:

try:

valid\_mask = dekadal\_df[col].notna()

if valid\_mask.sum() < 2:

continue

x\_vals = dekadal\_numeric[valid\_mask]

y\_vals = dekadal\_df[col][valid\_mask].values

if len(x\_vals) >= 4:

cs = CubicSpline(x\_vals, y\_vals, bc\_type='natural')

interpolated = cs(daily\_numeric)

else:

interpolated = np.interp(daily\_numeric, x\_vals, y\_vals)

interpolated = np.maximum(interpolated, 0)

# Round interpolated values to 2 decimal places

interpolated = np.round(interpolated, 2)

daily\_df[col] = interpolated

successful\_farms.append(col)

except Exception as e:

continue

print(f" {farm\_type}: ✓ {len(successful\_farms)} farms interpolated")

return daily\_df

def calculate\_daily\_averages(daily\_dfs):

"""Calculate daily average ET values for each farm type."""

avg\_df = pd.DataFrame()

for farm\_type, daily\_df in daily\_dfs.items():

if not daily\_df.empty:

daily\_avg = daily\_df.mean(axis=1, skipna=True)

avg\_df[farm\_type] = daily\_avg

print(f" {farm\_type}: {daily\_avg.min():.2f} - {daily\_avg.max():.2f} mm/day")

return avg\_df

def plot\_et\_comparison(avg\_df, period\_name="Analysis Period", save\_png=True):

"""Create comparison plot of ET trends across farm types."""

plt.figure(figsize=(14, 8))

colors = {'TPR': '#2E86C1', 'DSR': '#28B463', 'AWD': '#F39C12'}

line\_styles = {'TPR': '-', 'DSR': '--', 'AWD': '-.'}

for farm\_type in avg\_df.columns:

plt.plot(avg\_df.index, avg\_df[farm\_type],

color=colors.get(farm\_type, 'black'),

linestyle=line\_styles.get(farm\_type, '-'),

linewidth=3,

label=f'{farm\_type} (Avg: {avg\_df[farm\_type].mean():.2f} mm/day)',

marker='o', markersize=5, alpha=0.8)

plt.title(f'Evapotranspiration (ET) Trends Comparison\n{period\_name}',

fontsize=16, fontweight='bold', pad=20)

plt.xlabel('Date', fontsize=12, fontweight='bold')

plt.ylabel('ET Value (mm/day)', fontsize=12, fontweight='bold')

plt.gca().xaxis.set\_major\_formatter(mdates.DateFormatter('%Y-%m-%d'))

plt.gca().xaxis.set\_major\_locator(mdates.WeekdayLocator(interval=2))

plt.xticks(rotation=45)

plt.grid(True, alpha=0.3, linestyle=':', linewidth=0.5)

plt.legend(loc='upper left', frameon=True, fancybox=True, shadow=True, fontsize=11)

# Statistics box

stats\_text = f"""Statistics Summary:

Total Days: {len(avg\_df)}

Date Range: {avg\_df.index.min().strftime('%Y-%m-%d')} to {avg\_df.index.max().strftime('%Y-%m-%d')}

Farm Type Comparison:"""

for farm\_type in avg\_df.columns:

stats\_text += f"\n {farm\_type}: {avg\_df[farm\_type].mean():.2f} ± {avg\_df[farm\_type].std():.2f} mm/day"

plt.text(0.02, 0.98, stats\_text, transform=plt.gca().transAxes,

verticalalignment='top', bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.8),

fontsize=9, fontfamily='monospace')

plt.tight\_layout()

# Save plot as PNG

if save\_png:

plot\_filename = 'ET\_Farm\_Types\_Comparison\_Nov2024\_Apr2025.png'

plt.savefig(plot\_filename, dpi=300, bbox\_inches='tight', facecolor='white', edgecolor='none')

print(f"✓ Plot saved as: {plot\_filename}")

plt.show()

def generate\_summary\_statistics(avg\_df, daily\_dfs):

"""Generate detailed summary statistics."""

summary\_stats = []

for farm\_type in avg\_df.columns:

if farm\_type in daily\_dfs:

daily\_df = daily\_dfs[farm\_type]

avg\_values = avg\_df[farm\_type]

stats = {

'Farm\_Type': farm\_type,

'Num\_Farms': len(daily\_df.columns),

'Days\_Analyzed': len(daily\_df),

'Mean\_ET': round(avg\_values.mean(), 3),

'Std\_ET': round(avg\_values.std(), 3),

'Min\_ET': round(avg\_values.min(), 3),

'Max\_ET': round(avg\_values.max(), 3),

'Total\_ET': round(avg\_values.sum(), 3),

'CV\_Percent': round((avg\_values.std() / avg\_values.mean()) \* 100, 2)

}

summary\_stats.append(stats)

return pd.DataFrame(summary\_stats)

def main():

"""Main analysis function"""

print("🌾 ET TREND COMPARISON ANALYSIS")

print("="\*60)

# Test file access

input\_file = test\_file\_access()

if input\_file is None:

return

period\_name = "November 2024 - April 2025"

try:

print("\n📊 LOADING DATA...")

farm\_tables = load\_farm\_type\_tables(input\_file)

if len(farm\_tables) == 0:

print("❌ No valid tables found")

return

print("\n🔄 APPLYING SPLINE INTERPOLATION...")

daily\_dfs = {}

for farm\_type, dekadal\_df in farm\_tables.items():

daily\_df = interpolate\_to\_daily(dekadal\_df, farm\_type)

if not daily\_df.empty:

daily\_dfs[farm\_type] = daily\_df

if len(daily\_dfs) == 0:

print("❌ No daily data generated")

return

print("\n📈 CALCULATING AVERAGES...")

avg\_df = calculate\_daily\_averages(daily\_dfs)

print("\n📋 SUMMARY STATISTICS:")

summary\_stats = generate\_summary\_statistics(avg\_df, daily\_dfs)

print(summary\_stats.to\_string(index=False))

print("\n📊 CREATING PLOT...")

plot\_et\_comparison(avg\_df, period\_name)

print("\n💾 SAVING RESULTS...")

# Save files

try:

avg\_df.reset\_index().to\_excel('nov\_april\_daily\_averages.xlsx', index=False)

summary\_stats.to\_excel('nov\_april\_statistics.xlsx', index=False)

with pd.ExcelWriter('nov\_april\_all\_farms.xlsx') as writer:

for farm\_type, daily\_df in daily\_dfs.items():

daily\_df.reset\_index().to\_excel(writer, sheet\_name=f'{farm\_type}\_Daily', index=False)

print("✓ Files saved successfully!")

except Exception as e:

print(f"⚠ Could not save files: {e}")

print("\n" + "="\*60)

print("🎉 ANALYSIS COMPLETED!")

print("="\*60)

# Key findings

print(f"\n🔍 KEY FINDINGS for {period\_name}:")

mean\_ets = {}

for farm\_type in avg\_df.columns:

mean\_et = avg\_df[farm\_type].mean()

mean\_ets[farm\_type] = mean\_et

print(f"• {farm\_type}: {mean\_et:.2f} mm/day")

if len(mean\_ets) > 1:

highest\_et = max(mean\_ets, key=mean\_ets.get)

lowest\_et = min(mean\_ets, key=mean\_ets.get)

print(f"\n🏆 Highest ET: {highest\_et} ({mean\_ets[highest\_et]:.2f} mm/day)")

print(f"🥉 Lowest ET: {lowest\_et} ({mean\_ets[lowest\_et]:.2f} mm/day)")

print(f"📏 Difference: {mean\_ets[highest\_et] - mean\_ets[lowest\_et]:.2f} mm/day")

except Exception as e:

print(f"❌ Error: {str(e)}")

import traceback

traceback.print\_exc()

# Run the analysis

if \_\_name\_\_ == "\_\_main\_\_":

main()

# If running in Jupyter/Colab, execute directly

main()