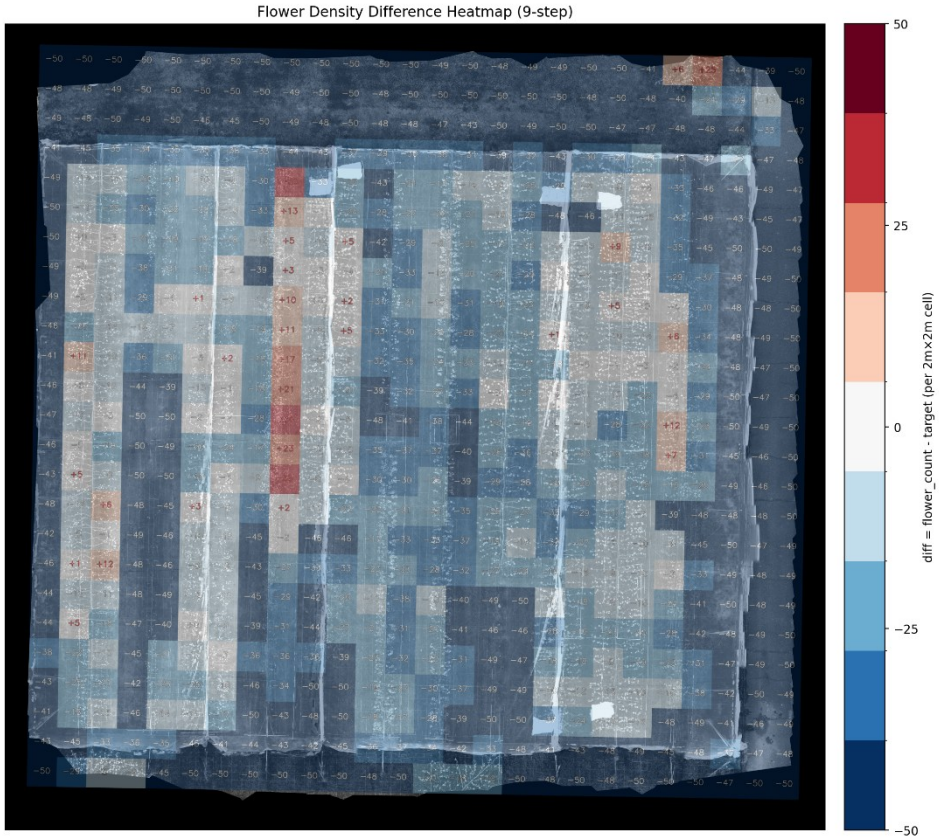
ドローン画像を用いた果樹の開花着果解析と営農管理ツールの連携による着果管理技術

～ニホンナシ花そう密度の見える化～



果樹ドローンコンソーシアム　編

令和４年度補正予算及び令和５年度当初予算「戦略的スマート農業技術の開発・改良」

改定履歴

版数	発行日	改定者	改定内容
第１版	2026 年 3 月 10 日	新潟食料農業大学 伊藤	初版発行

目次

免責事項

・利用者が本マニュアルに記載された技術を利用したこと、または利用できなかったことに起因して生じたいかなる結果についても、本マニュアルの作成者は一切の責任を負わないものとします。

・本マニュアルに記載したセンシング技術は、新潟県下越地方の園地において、ニホンナシの樹体ジョイント仕立てを対象に、Mavic 3M（DJI 社）の RGB カメラを用いて実施した事例に基づくものです。本技術の適用精度は、実施条件によって変動する可能性があることに留意ください。

第１章　はじめに

１．本マニュアルの目的と活用方法

ニホンナシでは、着果管理（受粉、摘果）に要する作業時間が多く、他作業に比べ機械化、省力化対応が遅れているため、省力化のボトルネックとなってきました。着果管理に手間がかかる原因は、①実を成らせる作業（花粉採取、受粉作業）に多大な労力を要すること、②実を減らす作業（摘果）にも作業時間を要することの 2 点です。実を減らす技術の省力化については、技術が未開発（摘果剤未登録、摘果ロボット未開発）ですが、既往の研究により、摘果作業時間には花数が影響する（川瀬ら 1996、多比良ら 1999）、摘果作業時間削減に花芽の除芽による開花数制限が有効（岩谷ら 2018、松本ら 2019）、自家和合性品種は花芽数制限で摘果作業の大幅省力化が可能（松本ら 2015）という知見が得られています。

以上のことから、着果管理省力化の近道は「シンプルに実を成らせすぎないこと！」です。つまり、必要数の花を開花させ、省力的かつ確実に着果させることが有効です。しかし、生産現場では広い果樹園の中で花芽数や花数をエリア単位で把握し、調整することは困難でした。

そこで本研究では、ドローン画像撮影、AI による花そうの認識計数をエリア単位で行い、スマートフォンの営農管理アプリとの連携により効率的な開花数制限を行うという、いわゆるデータ駆動型着果管理技術を開発しました（図 1）。本技術を活用することで、果樹園内の花そう数をエリア単位で把握し、摘花作業が必要なエリアを花そう密度ヒートマップにより一目で判断することができます。その結果、摘花作業時間の削減や省労力化が可能になります。

本マニュアルでは、花そう密度ヒートマップの作成方法（図 1②～③）を中心に紹介します。ドローンによるほ場全面を自動飛行で撮影する方法（図 1①）については、「ドローンの自動飛行による果樹の送風受粉運用マニュアル」および「ドローンセンシングを活用したリンゴ褐斑病対策技術 ①リンゴ褐斑病ハザードマップの作成方法」を参照してください。

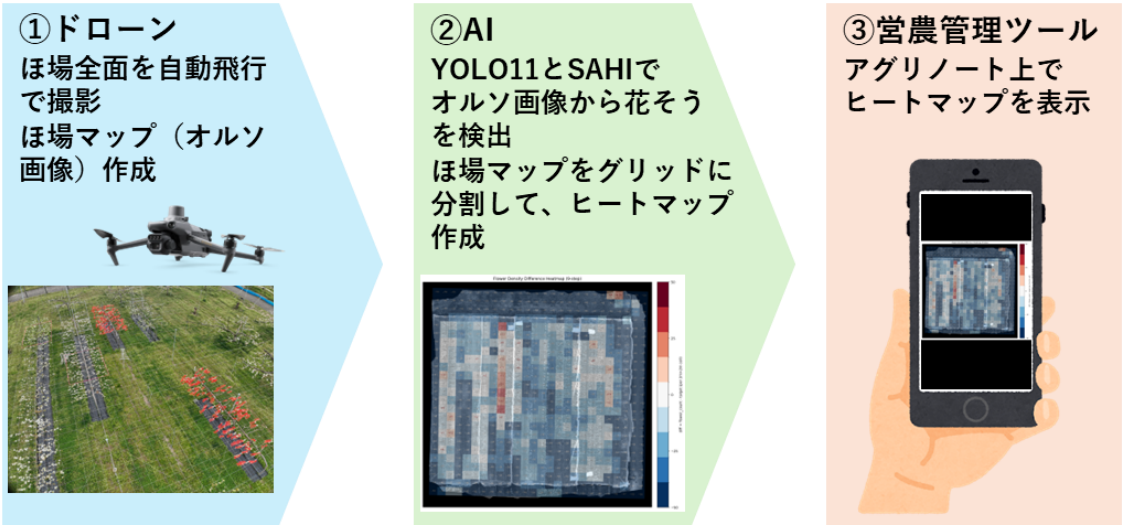


図１-１　ドローン画像を活用したニホンナシ摘花作業省労力化技術のイメージ図

① ドローン：ほ場全面を自動飛行で撮影、ほ場マップ（オルソ画像）作成

② AI：YOLO11 と SAHI でオルソ画像から花そうを検出、圃場マップをグリッドに分け、ヒートマップを作成

③ 営農管理ツール：アグリノート上でヒートマップを閲覧

２．ドローン画像を用いた果樹の開花着果解析と営農管理ツールの連携による着果管理技術の特徴

（１）活用・特徴

・ニホンナシの花そう密度の多寡を地図上に表示した「ニホンナシ花そう密度ヒートマップ」を作成できます。
・花そうの多い地点を把握し、重点的かつスムーズに摘花作業を実施することができます。
・ヒートマップは画像データとして表示・印刷ができる他、営農管理ツールなどを使用することでスマートフォンなどの携帯端末でも表示することができます。
・本マニュアルで使用する AI はオープンソースを使用しています。

（２）留意点

・除去した花そうが地面に落ちていたり、地面に白い石などが多く存在したりする条件では、適切な評価が困難となります。
・ヒートマップの作成にはドローンやオルソ画像作成ソフト、AI の実行環境等、特殊な機器やソフトウェアが必要となります。

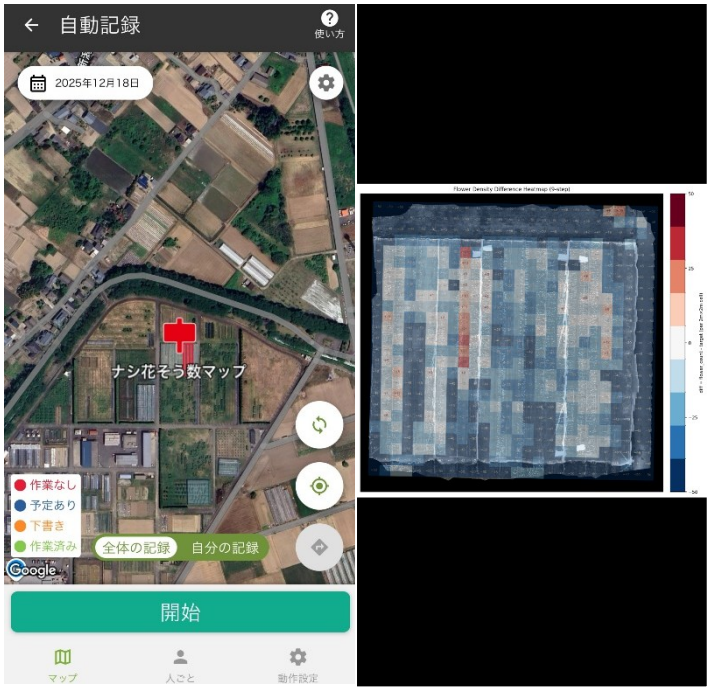


図１−２　営農管理ツール「アグリノート」にヒートマップをアップロードした例
左：看板機能、右：ヒートマップ

３．「ニホンナシ花そう密度ヒートマップ」の主な作成手順

（１）ドローンでほ場全体を空撮し、ほ場マップ（オルソ画像）を作成

・ドローンの RGB カメラを用いてほ場全体を撮影（オルソ画像撮影モード）

・撮影高度は地上高 16m 程度（本マニュアルでは 15.7m で飛行）

・撮影後に DJI Terra で Geo TIFF 形式のオルソ画像を作成

・具体的な撮影方法については、「ドローンの自動飛行による果樹の送風受粉運用マニュアル」および「ドローンセンシングを活用したリンゴ褐斑病対策技術 ①リンゴ褐斑病ハザードマップの作成方法」を参照してください。

（２）AI で画像解析し、ヒートマップを作成

・オルソ画像を画像処理（正規化、ガンマ補正）して物体検出に適した形式に変換

・物体検出モデル（YOLO11）と小さな物体の検出精度を高める画像分割処理（SAHI : Slicing Aided Hyper Inference）を用いて、オルソ画像から花そうを検出

・画像解析により花そうの位置情報を取得し、オルソ画像上で花そう密度を計算

・グリッドごとの花そう数と目標花そう数の差を求め、ヒートマップを作成

（３）営農管理ツールでヒートマップを閲覧

・営農管理ツール「アグリノート」の看板機能を活用し、パソコンやスマートフォン上でヒートマップを表示します。

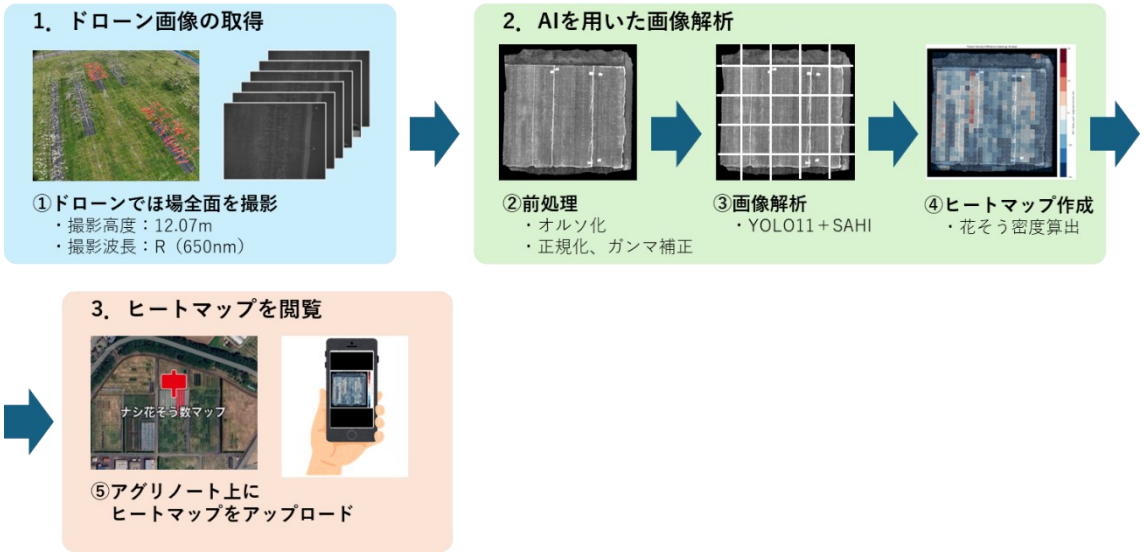


図１−３　「ニホンナシ花そう密度ヒートマップ」の主な作業手順

表１-１　主な使用機器・サービス

項目	詳細・推奨仕様	備考
1. ドローン関係		
カメラ	マルチスペクトルカメラ	・有効画素数5MP以上を推奨 ・有効画素数5MP以上を推奨マニュアルではMavic3Mのマルチスペクトルカメラを使用
機体機能	高精度な位置測位機能（GNSS、RTK）	・RTK機能による測位を強く推奨 ・マニュアルではD-RTK2（DJI社）を使用
	自動飛行機能	・広範囲の効率的なデータ取得に必要 ・マニュアルではDJI Pilot2（DJI社）を使用
2. 画像処理関係		
物体検出モデル	YOLO11（Ultralytics社）	・ドローン画像から「ナシの花そう」の検出に必要 ・マニュアルでは、独自のデータセットでYOLO11を転移学習したモデルを使用
ハードウェア／サービス	NVIDIA製GPU搭載PCまたはクラウドサービス	・物体検出モデル（YOLO11）を実行可能な環境が必要 ・マニュアルでは、NVIDIA GeForce RTX 4090（24GB）を搭載したパソコンを使用
ストレージ	10GB/ha程度の空き容量	・画像データの保存や画像処理に必要 ・マニュアルではローカルのストレージを使用したが、Googleドライブ（Google社）なども
ソフトウェア	SfM（Structure from Motion）ソフト	・ドローンで撮影した画像からほ場マップ（オルソ画像）を作成する際に使用 ・マニュアルではDJI Terra（DJI社）を使用

４.「ニホンナシ花そう密度ヒートマップ」の作成者

ヒートマップの作成にはドローンを用いた空撮、Python を用いた AI 画像解析やヒートマップの作成等、様々な機器やソフトウェアを使用します。このため、マップの作成者は関連する知識を有する専門の企業（ドローン作業受託企業、ソフトウェア企業）や指導機関（都道府県、JA など）を想定しています。

具体的なマップの作製手順については、本マニュアル第２章で紹介します。

５.「ニホンナシ花そう密度ヒートマップ」の利用者

ニホンナシの花そう密度を面的に把握したい、生産者や指導者（JA、県等）の利用を想定しています。実施者自らドローン画像を撮影する場合は小規模な面積から利用可能ですが、業者に委託する場合は数 ha 規模での利用が現実的と思われます。

第2章　AIを用いた画像解析

１．概要紹介

ここでは、AI を用いた画像解析の標準的な手順を紹介します。

画像解析の実行には、NVIDIA 社製 GPU を搭載した高性能な計算リソース（PC またはクラウド環境）や、Python プログラムに関する専門知識が不可欠です。したがって、本手順の実施主体を、AI 解析に対応した民間企業、専門の委託業者、または指導機関（JA、県など）と想定し、基本的な操作や用語の詳細な説明は割愛します。

２．使用する AI（物体検出モデル）

2023 年にイギリスの Ultralytics が公開した YOLO11（You Only Look Once 11）を使用します。YOLO11 は AI 画像解析フレームワークの一つで、画像内の対象物を自動で検出し、その位置を示す物体検出モデルです。農業・医療・産業分野など幅広い領域で利用されているモデルの一つです。また、オープンソースとして公開されており、研究用・業務用のどちらでも無償で利用できます。

本マニュアルでは、COCO データセット（val2017）で学習済みの YOLO11n を基に、ニホンナシの花そうを識別できるよう、独自のデータセットを用いて再調整（ファインチューニング）したモデルを使用しています。

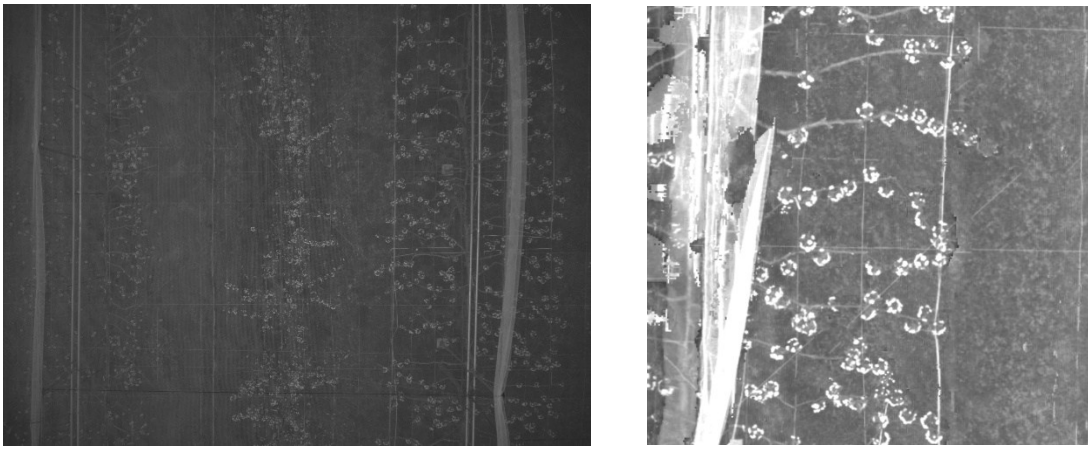


図２－１　独自のデータセットに含まれる画像（例）

左：ドローン（Mavic3M, DJI 社）のマルチスペクトルカメラの R 波長で撮影、

右：ドローンで撮影した画像をオルソ化・画像処理（正規化、ガンマ補正）した。

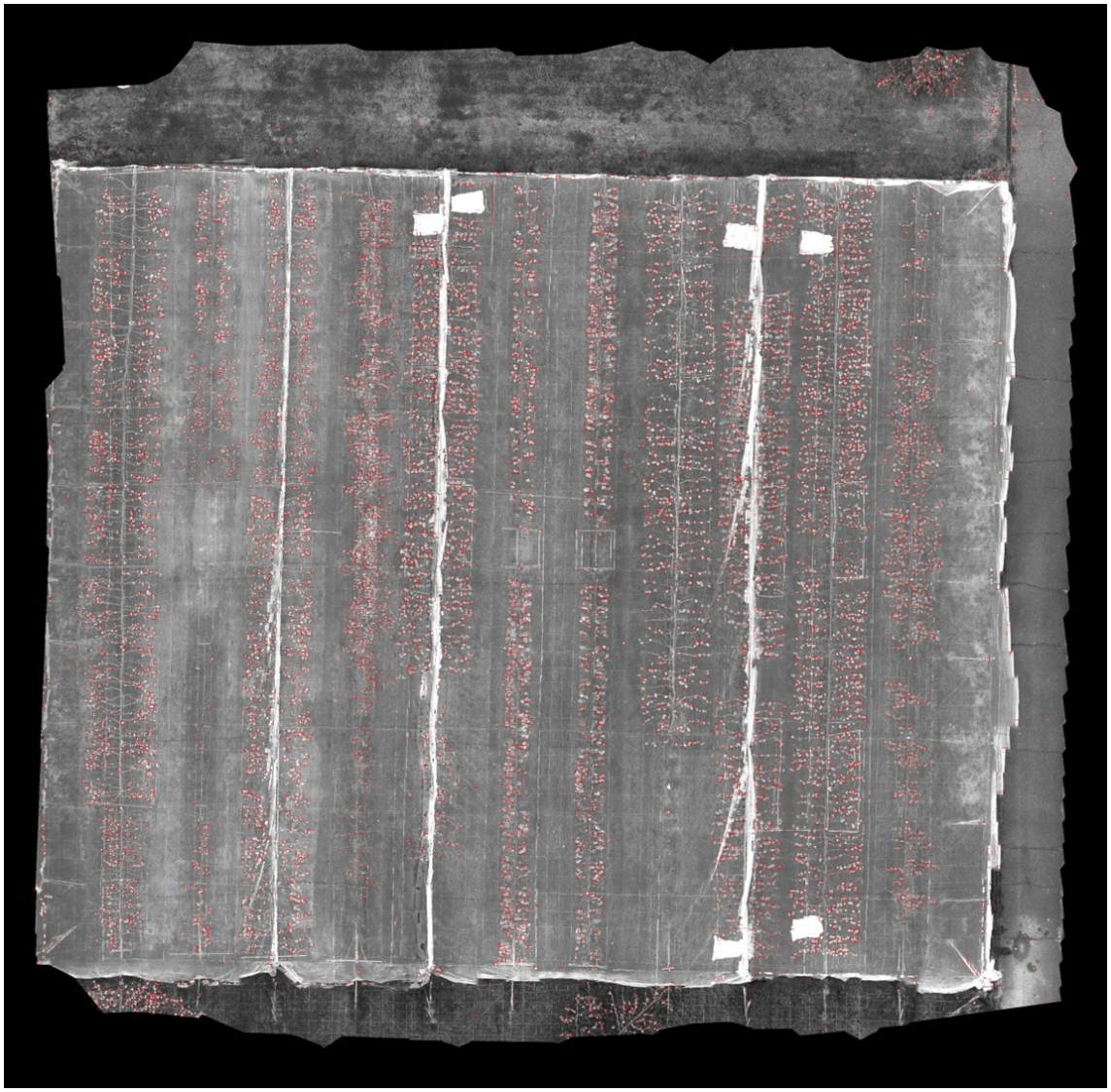


図２-２　オルソ画像に対する検出結果

図内の赤枠線はモデルが「花そう」と判別した場所を示す。

３．画像の前処理

（１）ほ場マップ（オルソ画像）の作成

DJI Terra を用いて、ドローンで連続撮影した画像から 1 枚のオルソ画像（GeoTIFF 形式）を作成します。

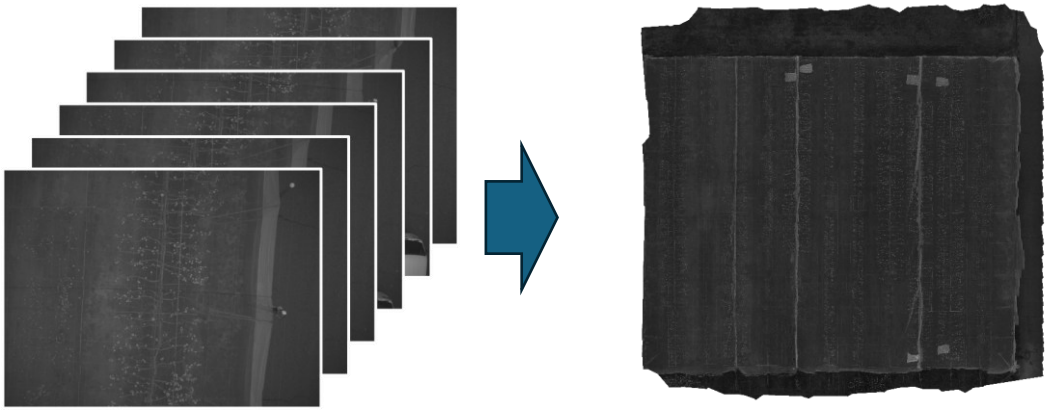


図２−３　オルソ画像化

左：元画像 2,592 × 1,944 ピクセル、右：オルソ画像 11,2828 × 11,096 ピクセル

（２）画像処理

DJI Terra により出力されるオルソ画像（GeoTIFF）は 16bit ですが、YOLO/SAHI は 8bit RGB が前提です。そのため、下記のコードにより GeoTIFF を、8bit RGB の一般的な画像形式（PNG）に変換して、AI や画像処理で使いやすくします。また、花そうと背景が明確ではなく検出しづらいため、ガンマ補正を行って花そうを判別しやすくします。

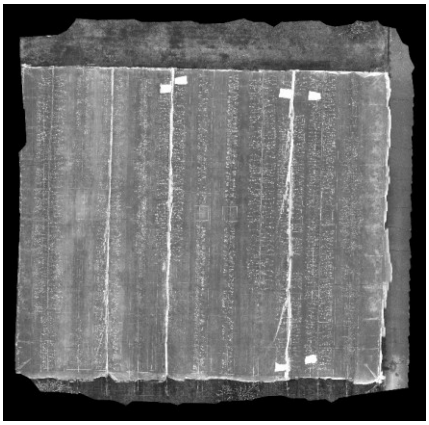


図２−４　正規化・ガンマ補正をしたオルソ画像

オルソ画像（GeoTIFF）をパーセンタイル正規化（2〜98%）、ガンマ補正 1.3 乗

下記のコード（Python）を実行すると、オルソ画像（GeoTIFF）をパーセンタイル正規化・ガンマ補正して 8bit PNG 形式に変換することができます。

```
# ライブラリのインポート
import rasterio  # 地図データ（GeoTIFF）を扱うためのライブラリ
import numpy as np  # 数値計算（行列）を高速に行うためのライブラリ
import cv2  # 画像処理（保存など）を行うためのライブラリ（OpenCV）

# 読み込む元の地図ファイルの場所
tif_path = "C:/Users/test/ortho.tif"
# 変換した後に保存する先の場所
png_path = "C:/Users/test/ortho_rgb_8bit.png"

# オルソ画像の読込と形状変換
with rasterio.open(tif_path) as src:
    img = src.read([1, 2, 3]).astype(np.float32)

# パーセンタイル正規化（各チャンネルを個別に正規化）
## 画像の中で「極端に暗い 2%」と「極端に明るい 2%」をノイズとして無視し、
## メインの 96%の範囲が「0〜255（画像として標準的な明るさ）」に収まるように調整
for c in range(3):  # R, G, B の 3 つの層に対して 1 つずつ処理
    p2, p98 = np.percentile(img[c], (2, 98))  # 正規化の強度（強め 2, 98、弱め 5, 95）
    img[c] = np.clip((img[c] - p2) / (p98 - p2) * 255, 0, 255) # 範囲を引き伸ばす

# ガンマ補正（明るさの微調整）
gamma = 1.3  # 補正の強さ（1.0 より大きいと明るくなる。1.2〜1.6 程度）
img = np.power(img / 255.0, gamma) * 255

# 画像形式の調整 & チャンネル順変更
img = img.transpose(1, 2, 0).astype(np.uint8)

# 画像の保存（加工したデータを png_path で指定した名前で保存）と書き出し
cv2.imwrite(png_path, img)
print("保存完了:", png_path)
```

✧ rasterio.open：オルソ画像を開きます。

✧ src.read([1,2,3])：画像から 1 番目（R：赤）、2 番目（G：緑）、3 番目（B：青）のチャ

ンネルを読み込みます。

- ✧ .astype(np.float32)：データの型を「整数」から「小数（浮動小数点数）」に変換します。この後の計算（割り算など）で、細かい数値が切り捨てられないようにするための重要な下準備です。
- ✧ 正規化（8bit 化）：オルソ画像は「16bit（0～65535 の値）」などで保存されていることが多く、そのまま PNG にすると真っ黒に見えたり、エラーになったりします。
- ✧ np.clip(..., 0, 255)：計算の結果、0 未満になったものは 0 に、255 を超えたものは 255 に固定します。
- ✧ img / 255.0：一旦 0.0～1.0 の範囲に縮小します。
- ✧ np.power(..., gamma)：1.3 乗の計算をします。これにより、暗い部分は維持しつつ、中間層をふんわりと明るくできます。
- ✧ transpose(1, 2, 0)：読み込んだ直後の形（色，縦，横）を変換後の形（縦，横，色）に並び替えます（多くの AI モデルは（縦，横，色）の順番でないと画像を認識できない）
- ✧ astype(np.uint8)：データを「0～255 の整数型（8bit）」に強制的に変換します。これで一般的な画像ファイルと同じ形式になります。
- ✧ cv2.imwrite：出来上がったデータを、指定した名前で PNG 画像として保存します。
- ✧ 黄色で塗ってあるパスやファイル名、値は、必要に応じて適宜修正してください。

（３）小さな物体の検出精度向上(SAHI)

ドローン画像からの検出には、YOLO11 に加え、2022 年にトルコの研究グループ OBSS が公開した SAHI（Slicing Aided Hyper Inference）を使用します。SAHI は、大きな画像をオーバーラップさせながら複数の小領域に分割し、それぞれを個別に解析することで、小さな物体の検出精度を向上させる前処理手法です。

本技術では、（１）（２）で作成・画像処理したオルソ画像を SAHI により画像をスライスしてモデル（YOLO11）に入力します。これにより、モデル入力時のリサイズによる解像度低下（情報の欠落）を回避し、ドローン画像からの花そう検出を可能にしています。

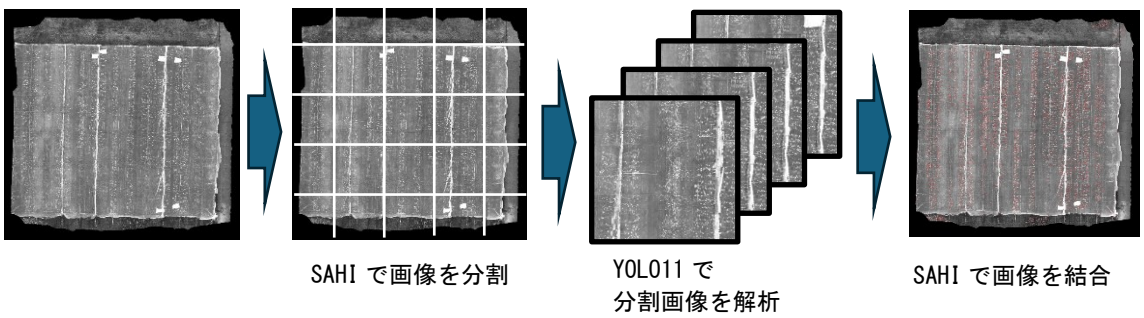


図２－５　SAHI と YOLO11 による画像処理の流れ

４．検出限界（理論値）

ドローンセンシングにおける検出限界は、主に「カメラの光学性能」と「AI モデルの特性」という二つの要素に依存します。

ドローン「DJI Mavic 3M」のマルチスペクトルカメラ（1/2.8 インチ CMOS、有効画素数 5MP、画像解像度 2592×1944 ピクセル、焦点距離 25mm 相当、視野角 61.2° ×48.10°）を用いた場合、撮影距離別の画素寸法(画像上の 1 ピクセルが示す実空間の大きさ）は以下のとおりです。

距離 1m：約 0.46 mm/pix
距離 5m：約 2.28 mm/pix
距離 10m：約 4.56 mm/pix

マニュアルでは、画素寸法 5.5 mm/pix（高度 12.07m）で撮影したドローン画像を使用して花そうの検出を実施しています。

５．プログラム（画像解析）の実行手順

解析に必要な学習済みモデルを配布しています。入手方法については、巻末の問い合わせ窓口にご相談ください。

本マニュアルではローカル環構の構築方法については触れません。Ultralytics 社の HP 等で YOLO11 および SAHI の実行環境を確認し、自身の環境にあったものを選択してください。

Ultralytics 社 HP 内　YOLO11 に関する情報
https://docs.ultralytics.com/ja/models/yolo11/
Ultralytics 社 HP 内　SAHI に関する情報
https://docs.ultralytics.com/ja/guides/sahi-tiled-inference/#benefits-of-sliced-inference

参考までに、本試験で使用したパソコンの仕様と主な環境を記載します。
PC の主な仕様
・OS　Windows 11 Home
・プロセッサ　14th Gen Intel(R) Core(TM) i7-14700KF (3.40 GHz)
・RAM　64.0 GB
・GPU　NVIDIA GeForce RTX 4090 (24 GB)
主な実行環境（仮想環境で実行しています）
・JupyterLab：4.3.4
・Python-3.9.20
・torch-2.0.1
・Ultralytics 8.3.78

（１）事前に用意するもの

・環境構築済みの画像解析用パソコン

・学習済みモデル：nashikasou_model.pt（注）

（注）実行プログラムおよび学習済みモデルの入手方法については、担当機関にご相談ください。

（２）YOLO11＋SAHI で花そうを検出、検出結果の可視化

```
# YOLO11 + SAHI で検出 ＋ bbox 可視化
from sahi import AutoDetectionModel　# SAHI 用のモデル読み込みツール
from sahi.predict import get_sliced_prediction　# 画像を分割して検出する関数
import os　# フォルダ作成などの OS 操作をするライブラリ

# ===== パス設定 =====
## 学習済みモデルが保存されている場所
model_path = "C:/Users/test/model/nashikasou_model.pt"
## 結果を保存するためのフォルダを指定
result_dir = "C:/Users/test/result/"
## もし結果を保存するためのフォルダが存在しなければ新しく作成する
os.makedirs(result_dir, exist_ok=True)

# ===== モデル読み込み =====
detection_model = AutoDetectionModel.from_pretrained(
    model_type="yolo11",　# 使用するモデルの種類（YOLO11）
    model_path=model_path,　# モデルファイルの指定
    confidence_threshold=0.2,　# 確信度 20%以上で「花そう」とみなす
    device="cuda:0"　# NVIDIA の GPU を使って高速計算する
)

# ===== SAHI による分割推論=====
## 巨大なオルソ画像（PNG）を 640×640 のサイズに切り分け、一つずつ物体検出を行う。
result = get_sliced_prediction(
    png_path,　# 入力する画像（前のステップで作った PNG 形式のオルソ画像）
    detection_model,　# 読み込んだモデル
    slice_height=640,　# 縦 640 ピクセルのタイルに分割
    slice_width=640,　# 横 640 ピクセルのタイルに分割
    overlap_height_ratio=0.4,　# タイル同士を 40%重ねる（縦）
    overlap_width_ratio=0.4　# タイル同士を 40%重ねる（横）
)

print("検出数:", len(result.object_prediction_list))　# 検出結果（花そう数の合計）表示
```

✧ confidence_threshold=0.2：この値を下げると、花そうを検出しやすくなるが誤検出も増えます。

✧ overlap：タイルの境界線上に「花そう」があった場合、分割されてしまうと検出に失敗します。そのため、わざと重なり（オーバーラップ）を作って全領域を確実にカバーします。

```
# 検出結果の四角い枠（bbox : Bounding Box）を OpenCV で可視化（太さ自由）
import cv2

img = cv2.imread(png_path)    # オルソ画像（PNG）を読み込み
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)  # 色の並びを BGR（青緑赤）から RGB（赤緑青）に入れ替える

# bbox 描画
for obj in result.object_prediction_list:
    bbox = obj.bbox   # 検出された 1 つ分の「枠」情報を取り出す
    x1, y1 = int(bbox.minx), int(bbox.miny)   # 左上の角（X 座標, Y 座標）
    x2, y2 = int(bbox.maxx), int(bbox.maxy)   # 左上の角（X 座標, Y 座標）

    cv2.rectangle(
        img,
        (x1, y1),              # 開始点（左上）
        (x2, y2),              # 終了点（右下）
        color=(255, 0, 0),   # 線の色（赤：R=255, G=0, B=0）
        thickness=1          # 線の太さ（数字を大きくすると太くなる）
    )

# 保存
out_path = os.path.join(result_dir,"kasou_bbox.png")
# 色の並びを RGB から BGR に戻して保存
cv2.imwrite(out_path, cv2.cvtColor(img, cv2.COLOR_RGB2BGR))

print("bbox 画像を保存しました:", out_path)
```

✧ int：OpenCV で図形を描くときは整数（ピクセル単位）である必要があるため、小数点以下の端数を切り捨てて整数に変換しています。

✧ cv2.rectangle：「四角を描け」という命令です。

- ✧ color :（赤，緑，青）の強さを 0〜255 で指定します。(255, 0, 0）で赤になります。
- ✧ thickness : ドローンの画像は非常に大きいので、もし線が細すぎて見えない場合は、ここを 3 や 5 に大きくすると見やすくなります。
- ✧ cv2.imwrite : 加工した画像を保存します。
- ✧ print : カッコ内の内容を表示

（３）花そうの緯度経度（bbox の中心座標）を取得＋サイズフィルタ

ここでは、AI が検出した花そうの中から、bbox のサイズによって「誤検出」を取り除き、正しいと思われる花そうのみ中心座標を特定します。具体的には、小さすぎる・大きすぎる bbox を除去します。

```
# bbox 中心の「ピクセル座標」を取得（サイズフィルタ付き）

centers = []              # 最終的に採用する「中心点」を入れる空のリスト
filtered_bboxes = []      # フィルタを通過した「四角い枠」を入れるリスト
all_bboxes = []           # フィルタをかける前の「すべての枠」を入れるリスト

# bbox サイズ制限（pixel² : ピクセル単位の面積）
MIN_AREA = 250       # これより小さいものを除外
MAX_AREA = 6000    # これより大きいものを除外

for obj in result.object_prediction_list:
    bbox = obj.bbox
    all_bboxes.append(bbox)     # まずは全ての検出結果を「フィルタ前」リストに保存

    # 四角い枠の 横幅(w) と 縦幅(h) を計算
    w = bbox.maxx - bbox.minx
    h = bbox.maxy - bbox.miny
    # 面積(area) を計算
    area = w * h

    #下記の条件でふるいにかける（フィルタリング）
    # 【面積】小さすぎる、または大きすぎる場合は除外
    if area < MIN_AREA or area > MAX_AREA:
        continue   # 以下の処理を飛ばして、次の「花そう」の判定へ移る
    # 【幅】横幅や縦幅が極端に短い（細長いノイズなど）は除外
    if w < 10 or h < 10:
        continue
    # 【幅】極端に長い（花そうにしては大きすぎる）ものは除外
    if w > 80 or h > 80:
        continue
```

```
        # 中心点（cx, cy）の計算
        cx = (bbox.minx + bbox.maxx) / 2
        cy = (bbox.miny + bbox.maxy) / 2

        centers.append([cx, cy])            # 中心点を保存
        filtered_bboxes.append(bbox)    # 合格した枠を保存

centers = np.array(centers)             # リストを計算しやすい「NumPy 配列」に変換

# 検出された花そうのうち、何個が合格したのかを表示
print("元の検出数:", len(all_bboxes))
print("サイズフィルタ後の bbox 数:", len(filtered_bboxes))
```

- ✧ result.object_prediction_list：AI が検出した全検出結果から 1 つずつ取り出します。
- ✧ 枠の右端（maxx）から左端（minx）を引いて横幅を出します。縦も同様です。
- ✧ continue：「このデータは不合格！次に行こう！」という指示です。

元の検出数とサイズフィルタ後の bbox 数が表示されるので、以下のことを確認してください。

- ・ 元の検出数が「（２）YOLO11＋SAHI で花そうを検出、検出結果の可視化」の検出数と一致しているか
- ・ フィルタリング前後の数字が極端だったり、同じだったりしないか

下記のコードによってフィルタリング前後の検出結果を描写して比べることができます。

```
# フィルタリング前後 bbox の比較表示

import matplotlib.pyplot as plt   #グラフや図を描写するライブラリ

# 元画像の読み込み
img_before = cv2.imread(png_path)
img_after  = cv2.imread(png_path)
```

```
# OpenCV 形式(BGR)から表示用形式(RGB)に変換
img_before = cv2.cvtColor(img_before, cv2.COLOR_BGR2RGB)
img_after  = cv2.cvtColor(img_after, cv2.COLOR_BGR2RGB)

# --- フィルタリング前（bbox：薄い赤） ---
for bbox in all_bboxes:
    x1, y1 = int(bbox.minx), int(bbox.miny)
    x2, y2 = int(bbox.maxx), int(bbox.maxy)
    cv2.rectangle(img_before, (x1, y1), (x2, y2), (255, 150, 150), 1)

# --- フィルタリング後（bbox：濃い青） ---
for bbox in filtered_bboxes:
    x1, y1 = int(bbox.minx), int(bbox.miny)
    x2, y2 = int(bbox.maxx), int(bbox.maxy)
    cv2.rectangle(img_after, (x1, y1), (x2, y2), (0, 0, 255), 1)

# 画面を左右に分割して表示（Matplotlib）
plt.figure(figsize=(14, 6))  # 横 14 インチ、縦 6 インチの大きな表示枠を作る

plt.subplot(1, 2, 1)          # 「1 行・2 列」の「1 番目」に配置
plt.imshow(img_before)        # フィルタ前の画像を表示
plt.title("Before filter")    # タイトルを付ける
plt.axis("off")               # 目盛り（ピクセル番号）を消す

plt.subplot(1, 2, 2)          # 「1 行・2 列」の「2 番目」に配置
plt.imshow(img_after)         # フィルタ後の画像を表示
plt.title("After filter")     # タイトルを付ける
plt.axis("off")               # 目盛りを消す

plt.show()                    # 実際に画面に表示する
```

✧ plt.subplot(1, 2, 1) ：画面を半分に割って左側に描くという命令です。

✧ 最後に plt.show() を実行することで、Jupyter Notebook などの画面上に 2 枚の画像が並んで表示されます。

フィルタリング前後の検出結果の画像が左右に表示されるので、以下の 2 点を確認してください。

1. 「Before」にはあるが「After」にはない枠
   ・それが「花そう以外（石、雑草、ビニールなど）」であれば、フィルタリングは成功
   ・　それが「花そう」であれば、MIN_AREA の設定が厳しすぎることになります。

2. 「After」に残っている変な枠
   ・　花そうではない大きな塊が青枠で囲まれていたら、MAX_AREA をもう少し小さく設定する必要があります。

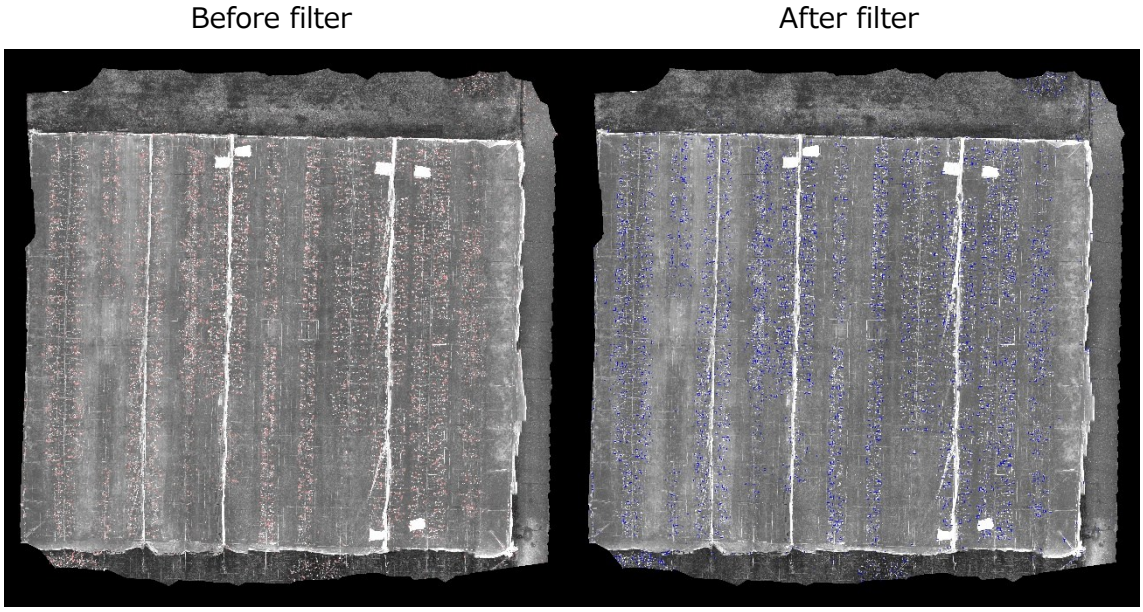


図２－６　フィルタリング前後の bbox 比較表示の例

さらに検出結果を精査する場合は、bbox サイズのヒストグラムを作成すると、明らかに小さすぎる・大きすぎる bbox の存在がわかります。

```
# bbox サイズを配列にまとめる（ヒストグラム用）

bbox_areas = []       # 面積（広さ）を記録するための空のリスト
bbox_widths = []      # 横幅を記録するための空のリスト
bbox_heights = []     # 縦幅を記録するための空のリスト
```

```
# bbox の横幅と縦幅から面積を計算
## フィルタ前の bbox（all_bboxes）かフィルタ後の bbox（filtered_bboxes）を選ぶ
for bbox in all_bboxes:              # フィルタ後の bbox を見る場合は、all→filtered に
    w = bbox.maxx - bbox.minx   # 右端の座標から左端の座標を引いて「横幅」を算出
    h = bbox.maxy - bbox.miny   # 下端の座標から上端の座標を引いて「縦幅」を算出
    area = w * h                # 「横 × 縦」で「面積（ピクセル数）」を算出

    bbox_widths.append(w)      # 横幅のリストに、上で計算した数値を書き込む
    bbox_heights.append(h)     # 縦幅のリストに、上で計算した数値を書き込む
    bbox_areas.append(area)    # 面積のリストに、上で計算した数値を書き込む

print("bbox サイズ配列を作成しました")
print("件数:", len(bbox_areas))

# bbox サイズのヒストグラム表示
plt.figure(figsize=(15, 4))  # 横 15 インチ、縦 4 インチの横長な「画用紙」を準備

# --- 1．面積のヒストグラム（左側）---
plt.subplot(1, 3, 1)                # 3 つ並べるうちの「1 番目（左）」を指定
plt.hist(bbox_areas, bins=100)  # 100 本の棒で面積の分布を描画
plt.title("bbox area (pixel^2)")  # グラフのタイトル（面積）
plt.xlabel("area")                  # 横軸のラベル（面積の値）
plt.ylabel("count")                 # 縦軸のラベル（見つかった個数）
plt.yscale("log")                   # 縦軸を「対数スケール」にする
# 面積のフィルタリング基準（250〜6000）をグラフに描写
plt.axvline(MIN_AREA, color="red", linestyle="--")
plt.axvline(MAX_AREA, color="red", linestyle="--")

# --- 2．幅のヒストグラム（中央）---
plt.subplot(1, 3, 2)                # 3 つ並べるうちの「2 番目（中央）」を指定
plt.hist(bbox_widths, bins=100)  # 横幅の分布を描画
plt.title("bbox width (pixel)")
plt.xlabel("width")
# 幅のフィルタリング基準（10〜80）をグラフに描写
plt.axvline(10, color="red", linestyle="--")
plt.axvline(80, color="red", linestyle="--")
```

```
# --- 3．高さのヒストグラム（右側）---
plt.subplot(1, 3, 3)                    # 3 つ並べるうちの「3 番目（右）」を指定
plt.hist(bbox_heights, bins=100)  # 縦幅の分布を描画
plt.title("bbox height (pixel)")
plt.xlabel("height")
# 高さのフィルタリング基準（10～80）をグラフに描写
plt.axvline(10, color="red", linestyle="--")
plt.axvline(80, color="red", linestyle="--")

plt.tight_layout()    # 隣り合うグラフの文字が重ならないように自動調整
plt.show()            # 画面に表示
```

- ✧ append ：「リストの最後に付け加える」という意味です。これで、検出された数だけデータが溜まっていきます。
- ✧ plt.yscale("log") ：普通のグラフだと、数が多いもの（例えば 1000 個）に隠れて、数が少ないもの（例えば 1 個や 2 個）は棒が低すぎて見えなくなります。対数（ログ）スケールにすると、「1 個しかない巨大な誤検出」や「数個しかない小さな誤検出」もグラフ上にしっかり浮かび上がって見えるようになります。
- ✧ plt.axvline：垂直な線（Vertical line）を引きます。これにより、「フィルタリングの境界線（赤線）の外側に、どれくらいデータがはみ出しているか」がわかります。

ヒストグラムについて、以下の 2 点を確認してください。

- **赤い線の外側に大きな山（データの塊）がある場合**：
  誤検出ではなく本物の「花そう」を除去している可能性があります。その時は、フィルタの数値（250～6000、10～80 など）をもっと広げる調整を検討してください。
- **赤い線の内側がスカスカで、外側にばかりデータがある場合**：
  AI が「花そう」ではないノイズばかりを拾っている可能性があります。

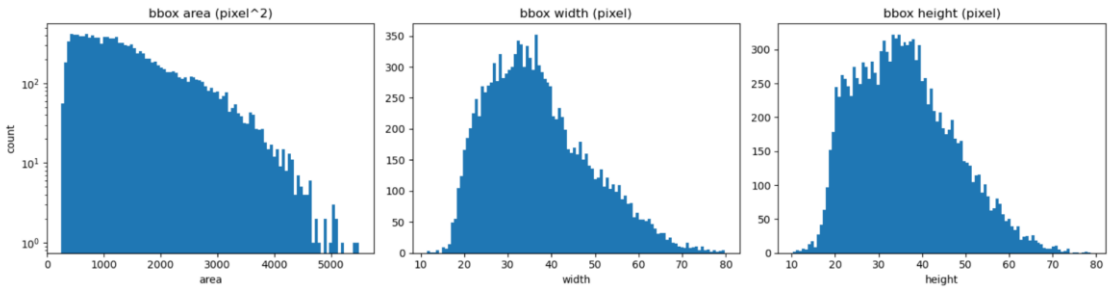


図２－７　bbox のヒストグラムの例

（４）bbox の位置情報をピクセルから UTM 座標系に変換

AI が検出した「花そう」について、「どのピクセルか」という情報を、オルソ画像（GeoTIFF）が持っている位置情報を利用して、UTM 座標系に変換します。この場合、出力される数値は「緯度経度（度、°）」ではなく、「ある起点からの距離（m）」を示します。

```
# bbox 中心ピクセル → UTM 座標系（単位：メートル）への変換

# 座標を記録するための「リスト」を準備
lats = []    # Y 座標（北方向への距離：Northing）が入る
lons = []   # X 座標（東方向への距離：Easting）が入る

# オルソ画像（GeoTIFF）の情報を読み込む
with rasterio.open(tif_path) as src:
    print("座標系の情報:", src.crs)
    print("EPSG コード:", src.crs.to_epsg())

# bbox の中心点を 1 つずつ「メートル単位」の座標に変換する
    for cx, cy in centers:              # bbox の中心ピクセル（x, y）を 1 つずつ取り出す
        lon, lat = src.xy(cy, cx)   # ピクセルを平面座標に変換

# 変換された「メートル座標」をリストに追加
        lats.append(lat)
        lons.append(lon)

# 結果を画面で確認
## 全部の座標を表示すると大変なので、最初の 5 個分だけを表示
print("平面直角座標（最初の 5 個）[単位:m]:")
for i in range(5):
    print(f"X(E): {lons[i]:.3f}, Y(N): {lats[i]:.3f}")
```

UTM 座標系とは

- カバー範囲：世界中を 6 度ずつの帯で分割
- 原点（南北）：赤道が 0 m
  ※新潟県（北緯 37～38 度付近）を m に直すと、赤道からおよそ 420 万 m
- 農業や森林管理の分野においては、距離や面積の計算やグリッドの分割などをする際に、緯度経度（度）よりも UTM 座標（メートル）の方が便利なことがあります。

- 座標系の情報が EPSG:32654 と出たら、WGS 84/ UTM zone 54N という内容です。UTM 座標系の第 54 帯は、新潟を含む東日本で一般的です。日本の大部分は 54 帯か 53 帯です。
- EPSG コードが EPSG:6691 と出たら、内容は JGD2011 / UTM zone 54N です。日本の測地系（JGD2011）の名前がついていますが、数値のルールは EPSG:32654 と一緒です。
- EPSG コードが EPSG:6676 や 6677 と出たら、UTM ではなく、日本独自の平面直角座標系（第 8 系や第 9 系）です。

✧ rasterio.open：画像ファイルを開き、その中に書き込まれている「この画像は UTM の第〇〇帯で、1pixel は何 cm か」といったメタデータ（位置情報ルール）を読み込みます。

✧ for cx, cy in centers：前のステップで絞り込んだ「花そう」の中心ピクセル（x, y）を 1 つずつ取り出します。

✧ src.xy(cy, cx)：ここが変換の核心です。オルソ画像内のピクセル位置を伝えると、ファイルのメタデータに基づいて、その場所の「平面座標（m）」を計算します。戻り値の lon は数学的な X（東向きの距離：Easting）、lat は Y（北向きの距離：Northing） に相当します。プログラミングの世界では「縦(y)＝行(row)」「横(x)＝列(col)」という順で指定する決まりがあるため、(cy, cx) という順になっています。

（４）座標系を回転させてヒートマップ用のグリッドを作る

圃場マップ（オルソ画像）を見ると、果樹の樹列がマップの垂直方向（南北方向）に対して傾いていることがあります。ヒートマップを作る際には、この傾きに合わせてグリッドを作る必要があります。そのため、圃場マップの中心点を定めて、どのくらい回転させるかを調整する必要があります。

```
# 座標系を回転させるための回転軸（中心点）と計算用パーツの準備

# ライブラリの読み込み
import math              # 数学計算（ラジアン変換など）
import numpy as np   # 数値計算（サイン・コサインの計算）
import rasterio          # GeoTIFF（地図データ）の情報を取得

# 角度の単位変換（度→rad）※時計回りに 6 度なら angle_deg = 6 のままで OK
angle_deg = 6                        # 回転させたい角度（時計回り）を指定
theta = math.radians(angle_deg)   # 「度」を「ラジアン」に変換

# GeoTIFF の中心（UTM 座標）を回転中心にする（固定）
with rasterio.open(tif_path) as src:
    b = src.bounds                          # 画像の四隅の座標（左・右・下・上）を取得
    x_center = (b.left + b.right) / 2       # 左右の平均値から中心の X 座標を決定
    y_center = (b.bottom + b.top) / 2       # 上下の平均値から中心の Y 座標を決定
ct, st = np.cos(theta), np.sin(theta)       # 回転行列の計算用に ct, st という変数を用意

# 回転軸の座標と計算に用いるラジアン値を表示
print("x_center,y_center =", x_center, y_center)
print("angle_deg =", angle_deg, " theta(rad) =", theta)
```

- ✧ 角度の単位変換式（度からラジアン）は、rad = deg × π ÷ 180 です。
- ✧ x_center：回転の中心となる横方向の住所（メートル）
- ✧ y_center：回転の中心となる縦方向の住所（メートル）
- ✧ ct：回転計算用の *cos* 変数。**c**os **t**heta（コサイン・シータ）の略
- ✧ st：回転計算用の *sin* 変数。**s**in **t**heta（サイン・シータ）の略
- ✧ プログラミングで座標回転を行う際、何度も np.cos(theta)や np.sin(theta)を計算させるのは非効率でコードも見づらいため、あらかじめ計算して短い名前の変数（ct, st）に保存しておきます。

```
# bbox 中心点の確認 → UTM 変換 → 回転 → グリッド集計（0 埋め込みまで）

# ライブラリの読み込み
import numpy as np      # 数値計算
import pandas as pd     # 表計算
import rasterio         # 地図データ読み込み
from rasterio.transform import xy as tf_xy   # ピクセル座標をメートル座標に変換

# グリッドサイズ（m）
grid_size_x = 2.0     # 横方向 2.0m 間隔（樹間や列間に合わせるとよい）
grid_size_y = 2.0     # 縦方向 2.0m 間隔（樹間や列間に合わせるとよい）

# 花そうの座標（centers） がメモリ上に存在するかチェックし、存在しない場合はエラーを返す。
if "centers" not in globals():
    raise ValueError("centers が見つかりません。bbox 中心のピクセル座標を取得してください。")

# 検出した花そうのピクセル座標を、扱いやすいように pandas の表形式にまとめる
bbox_df = pd.DataFrame({
    "x_c_px": centers[:, 0],    # 中心点のピクセル X
    "y_c_px": centers[:, 1],    # 中心点のピクセル Y
})

# ピクセル座標から UTM 座標への変換
with rasterio.open(tif_path) as src:
    xs, ys = tf_xy(             # ピクセル座標から UTM 座標に変換
        src.transform,
        bbox_df["y_c_px"].values,
        bbox_df["x_c_px"].values,
        offset="center"
    )
# 変換された UTM 座標を表の中に新しい列として追加
bbox_df["x_utm"] = np.array(xs)
bbox_df["y_utm"] = np.array(ys)
```

```
# UTM 座標の回転計算（画像の中心を軸にして座標を回す）
# 回転の軸（中心点）からそれぞれの花そうがどれくらい離れているか（差分）を計算
dx = bbox_df["x_utm"].values - x_center
dy = bbox_df["y_utm"].values - y_center
# 回転行列の計算
bbox_df["x_rot"] = ct * dx + st * dy
bbox_df["y_rot"] = -st * dx + ct * dy

# グリッド（方眼）の作成
# 回転後の座標において、一番左下（x_min)、一番下端（y_min）を見つけて、全体の外枠を決める
margin = 0.01    # 境界に乗る点があるときは 0.5 くらいに上げると安定します
x_min = bbox_df["x_rot"].min() - margin
y_min = bbox_df["y_rot"].min() - margin
x_max = bbox_df["x_rot"].max() + margin
y_max = bbox_df["y_rot"].max() + margin
print("rot 範囲:", (x_min, x_max, y_min, y_max))

# グリッド番号を割り当て（floor）
# （回転後の座標ー基準点を 2.0m で割って、小数点を切り捨てる（np.floor）ことで、その花そうが「何番目のマス目（グリッド番号）」かを割り振る
bbox_df["grid_x"] = np.floor((bbox_df["x_rot"] - x_min) / grid_size_x).astype(int)
bbox_df["grid_y"] = np.floor((bbox_df["y_rot"] - y_min) / grid_size_y).astype(int)

# 念のためグリッド番号にマイナスが出ていないか確認（マイナス時は警告）
if (bbox_df["grid_x"] < 0).any() or (bbox_df["grid_y"] < 0).any():
    print("⚠️ 警告: 負の grid index が出ています。margin を増やすか式を確認してください。")

# グリッドごとに花そう数（bbox 数）を集計
# 同じグリッド番号を持つ花そう（bbox）が何個あるかをカウントする
grid_counts = (
    bbox_df.groupby(["grid_x", "grid_y"])
    .size()
    .reset_index(name="flower_count")
)
```

```
# 0 埋め用：全グリッドを作る
# 花そうがある場所だけ数えると、花そうがない場所がデータから消えてしまう
max_gx = int(bbox_df["grid_x"].max())  # 横軸方向の最大グリッド数を確認
max_gy = int(bbox_df["grid_y"].max())  # 縦軸方向の最大グリッド数を確認

# ほ場全体の全てのグリッドのリストを作る
# 0 番から最大番までのグリッドのリストを生成
all_grids = pd.MultiIndex.from_product(
    [range(max_gx + 1), range(max_gy + 1)],
    names=["grid_x", "grid_y"]
).to_frame(index=False)

# グリッドリストに、花そうがあったグリッドの集計データ（花そう数）を重ねる
grid_counts = all_grids.merge(grid_counts, on=["grid_x", "grid_y"], how="left")
# 重ね合わせた結果、データが無かった（花そうがなかった）場所に 0 を記入
grid_counts["flower_count"] = grid_counts["flower_count"].fillna(0).astype(int)

# 最終確認
# ①元々の花そう数
print("bbox 総数:", len(bbox_df))
# ②グリッドごとに合計しなおした数（①と一致すれば計算ミスなし）
print("grid_counts flower_count 合計:", int(grid_counts["flower_count"].sum()))
# ③ほ場全体のグリッドの総数
print("全グリッド数(0 含む):", len(grid_counts))
```

- ✧ grid_size：ほ場を何メートル四方で区切るかを決めます。ここでは「2m×2m」のグリッドを作ろうとしています。
- ✧ tf_xy：画像上のピクセル番地を、UTM 座標系のメートル単位に一括変換します。
- ✧ margin: 計算ミスでデータが枠からはみ出さないよう、少しだけ外側に境界線を引きます。
- ✧ flower_count：そのグリッドに存在する花そうの数です
- ✧ all_grids：ほ場全体の全てのグリッドリストを作ります。
- ✧ merge & fillna(0)：全てのグリッドリストに集計結果を合体させ、花そうが見つからなかったグリッドには「0」を書き込みます。

「実際の画像マップの上にグリッドと検出した花そうを描き込んで、自分の目で確かめる（可視化する）」という下記のコードを実行すると、これまでの計算が正しかったかをより詳細に確認することができます。

```
#【検証】grid_counts と bbox_df の一致＋ 指定セルの画像保存

# ライブラリの読み込み
import os
import cv2
import numpy as np
import pandas as pd
import rasterio

# 保存先の設定（もし指定したフォルダがなければ新しく作成）
out_dir = "C:/Users/test/result"
os.makedirs(out_dir, exist_ok=True)

# --- １．全体一致チェック（ここは必ず一致するはず） ---
# グリッドごとの集計表（grid_counts）と花そうごとのリスト（bbox_df）の数が矛盾なく一致しているかをテストする
#花そうのリスト（bbox_df）をグリッドごとに数え、bbox_count という名前の表を作る
bbox_count = bbox_df.groupby(["grid_x","grid_y"]).size().reset_index(name="bbox_count")
# 集計済みの数（flower_count）と bbox_count の差を求めて、誤差を計算する
chk = grid_counts.merge(bbox_count, on=["grid_x","grid_y"], how="left")
chk["bbox_count"] = chk["bbox_count"].fillna(0).astype(int)
# 計算が合っていれば誤差（err）はすべて 0 になる
chk["err"] = chk["flower_count"] - chk["bbox_count"]

# 計算結果を表示する
print("一致率(err==0):", (chk["err"]==0).mean())  # 一致率が 1.0（100%）なら OK
print("MAE:", chk["err"].abs().mean())            # MAE は誤差の平均(0.0 なら OK)
print("±1 以内割合:", (chk["err"].abs()<=1).mean())
print("±2 以内割合:", (chk["err"].abs()<=2).mean())
```

```
# 誤差があった場合のために、どのグリッドの誤差が大きかったのかを示す
print("¥n 誤差が大きい上位:")
print(chk.loc[chk["err"].abs().sort_values(ascending=False).index][["grid_x","grid_y","flower_count","bbox_count","err"]].head(10))

# --- ２．指定した特定のグリッドを可視化して保存 ---
gx_t, gy_t = 24, 23    # 横 24 番、縦 23 番のグリッドを指定（番号は変更可能）

# グリッドの回転座標系での矩形範囲（グリッドの四隅の座標を計算）
x1r = x_min + gx_t * grid_size_x
x2r = x_min + (gx_t + 1) * grid_size_x
y1r = y_min + gy_t * grid_size_y
y2r = y_min + (gy_t + 1) * grid_size_y

# 回転させた後の座標
corners_rot = [(x1r,y1r),(x2r,y1r),(x2r,y2r),(x1r,y2r)]

# 回転座標 -> UTM 座標に戻す -> ピクセル座標に戻す
## 現実の画像マップは斜めのままなので、パソコン上で水平にした座標（xr, yr）を、斜めの画像に合わせるために、逆向きに回転させて戻す（元の UTM 座標に戻す）
## さらに UTM 座標をピクセル座標に変換
with rasterio.open(tif_path) as src:
    poly_px = []
    for xr, yr in corners_rot:
        x = x_center + (ct * xr - st * yr)
        y = y_center + (st * xr + ct * yr)
        row, col = src.index(x, y)
        poly_px.append([int(col), int(row)])
poly_px = np.array(poly_px, dtype=np.int32)

# 指定したグリッドに入っている花そう（bbox）のデータだけ抽出
inside = bbox_df[(bbox_df["grid_x"]==gx_t) & (bbox_df["grid_y"]==gy_t)].copy()

# 背景：元のドローン画像（PNG）を読み込み、色を正しく設定して描画の背景にする
img = cv2.imread(png_path)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
```

```
# 枠線：指定したグリッドの四隅を線で結んで黄色の太い枠線（ポリゴン）を描画
cv2.polylines(img, [poly_px], isClosed=True, color=(255,255,0), thickness=4)

# 丸：指定したグリッドの中にある花そうの場所に、緑色の丸（ドット）を描画
for _, r in inside.iterrows():
    cx, cy = int(round(r["x_c_px"])), int(round(r["y_c_px"]))
    cv2.circle(img, (cx, cy), 6, (0,255,0), -1)

# 文字情報：画像の左上に、グリッド番号、花そう数、誤差を黄色の文字で記入
flower = int(grid_counts[(grid_counts["grid_x"]==gx_t) & (grid_counts["grid_y"]==gy_t)]["flower_count"].iloc[0])
bboxn  = len(inside)
txt = f"(gx,gy)=({gx_t},{gy_t}) flower_count={flower} bbox_count={bboxn} err={flower-bboxn}"
cv2.putText(img, txt, (50, 80), cv2.FONT_HERSHEY_SIMPLEX, 2.0, (255,255,0), 5, cv2.LINE_AA)

# 全体保存（描画が完了した巨大な全体画像を PNG ファイルとして保存）
out_all = os.path.join(out_dir, f"verify_cell_gx{gx_t}_gy{gy_t}_ALL.png")
cv2.imwrite(out_all, cv2.cvtColor(img, cv2.COLOR_RGB2BGR))

# ズーム（指定したグリッドの周辺だけズームする）
xs = poly_px[:,0]; ys = poly_px[:,1]
pad = 250    # グリッドの周囲に 250 ピクセルの余白（pad）を持たせる
x1 = max(xs.min()-pad, 0); x2 = min(xs.max()+pad, img.shape[1]-1)
y1 = max(ys.min()-pad, 0); y2 = min(ys.max()+pad, img.shape[0]-1)
zoom = img[y1:y2, x1:x2].copy()
# ズーム保存（指定したグリッドの周辺で切り出して PNG ファイルとして保存）
out_zoom = os.path.join(out_dir, f"verify_cell_gx{gx_t}_gy{gy_t}_ZOOM.png")
cv2.imwrite(out_zoom, cv2.cvtColor(zoom, cv2.COLOR_RGB2BGR))

# 指定したグリッド内の花そうの座標データを CSV 保存
out_csv = os.path.join(out_dir, f"verify_cell_gx{gx_t}_gy{gy_t}_inside_points.csv")
inside.to_csv(out_csv, index=False)
```

```
# どこに何を保存したかを画面に表示
print("保存:", out_all)
print("保存:", out_zoom)
print("保存:", out_csv)
print("内容:", txt)
```

✧　src.index(x, y)：戻した UTM 座標（メートル）を、さらに「画像の何ピクセル目か」に変換します。

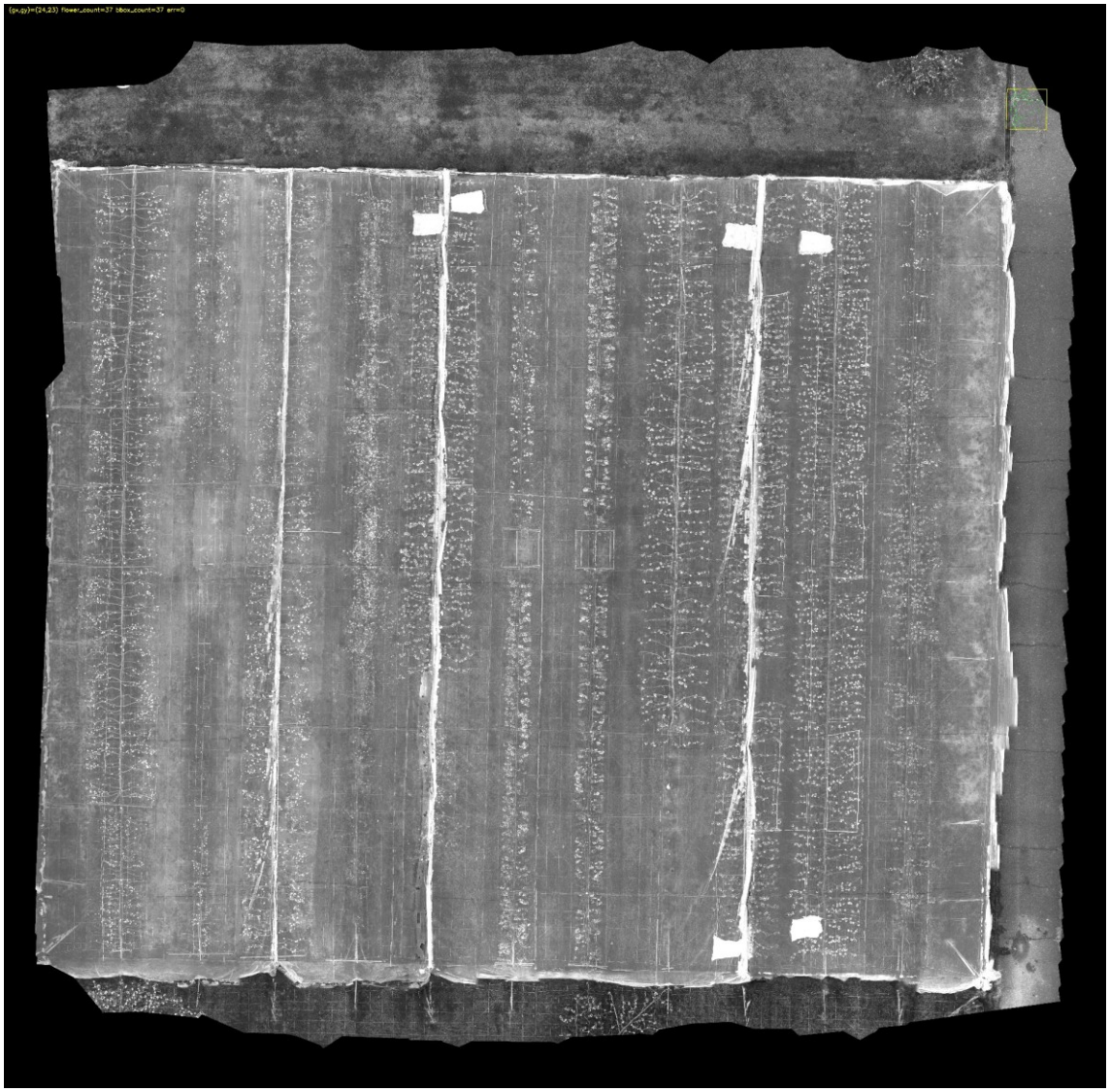


図２－８　全体保存の例

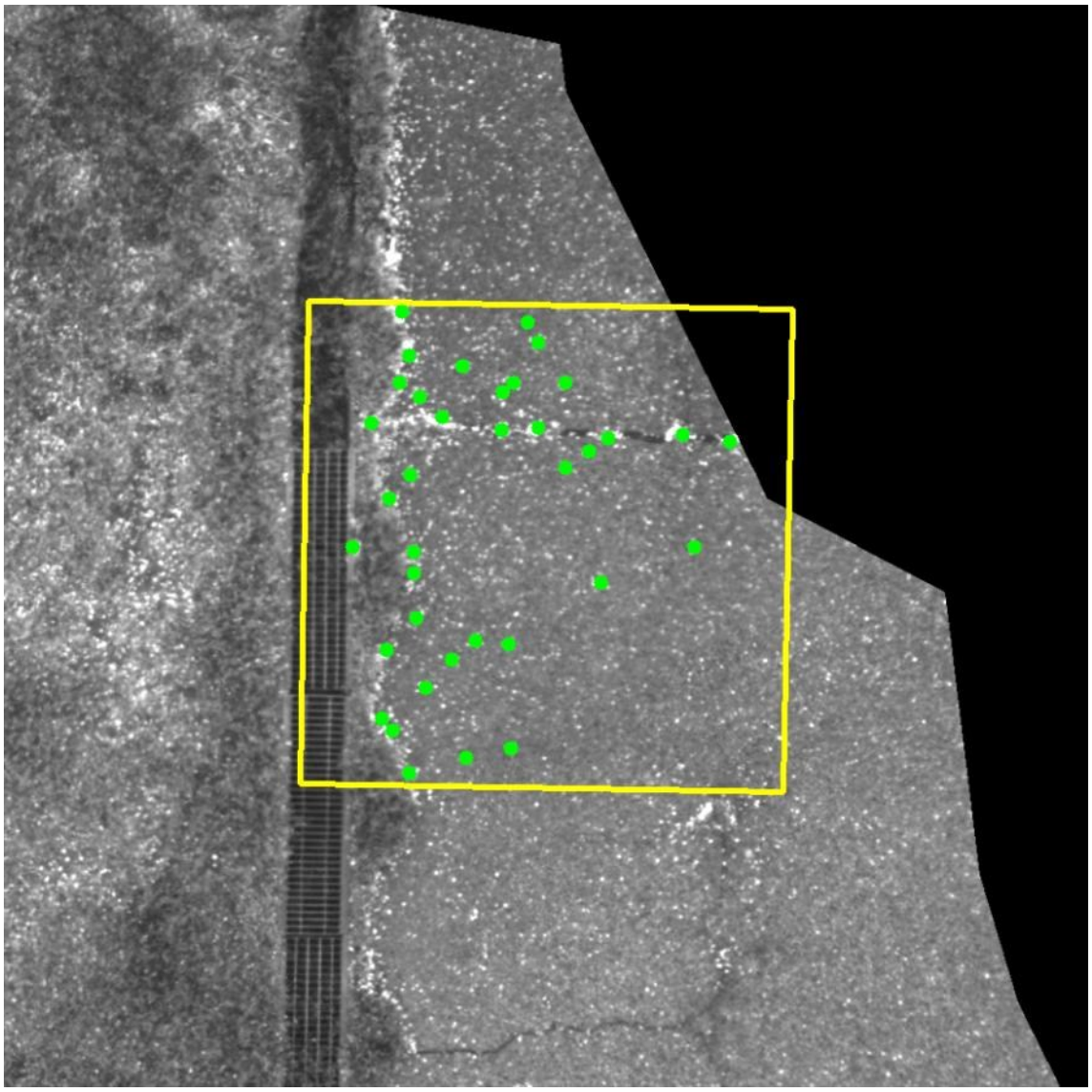


図２－９　ズーム保存の例（黄色の枠がグリッドの境界、緑色の丸が bbox の中心）

（５）検出した花そう数と目標花そう数との差を求める

ここでは目標花そう数を「1m² あたり 12.5 個」と定めて計算しています。この数値を書き換えることで、目標値を自由に変更できます。AI で検出した花そう数から目標花そう数を引くことで、差分を求めてグリッドの花そう数の多寡を判断します。

（検出した花そう数）－（目標花そう数）＝……

- プラスの値：目標より花そうが多い区画（摘花が必要）
- マイナスの値：目標より花そうが少ない区画（摘花は不要だが、着果率を上げたい）
- 0 に近い値：目標通りの順調な区画

```
# 面積基準（個 / ㎡）から diff を計算

# ① 面積あたりの目標花数（固定）
target_density = 12.5          # ← 1 ㎡あたり 12.5 個の花（必要に応じて調整する）

# ② グリッド 1 マスの面積（㎡）
grid_area = grid_size_x * grid_size_y

# ③ グリッド 1 つあたりの目標花数
target_flower_count = target_density * grid_area

# flower_count（実際に検出した花そう数）のデータがあるか確認（なければエラー）
if "flower_count" not in grid_counts.columns:
    raise ValueError("grid_counts に flower_count 列がありません。セル 6-2/6-3 を先に実行してください。")

# ④ 差分を計算（実際に検出した花そう数から目標花そう数を引き算）
grid_counts["diff"] = grid_counts["flower_count"] - target_flower_count

# 計算結果の出力と確認
print("grid_counts columns:", grid_counts.columns.tolist())
print("diff の確認:", grid_counts["diff"].head())
print(f"グリッド面積: {grid_area:.2f} ㎡")
print(f"理想花数（1 マス）: {target_flower_count:.2f} 個")
```

```
# ⑤ 確認（完成した表の最初の 5 行を表示して、列が正しく並んでいるか確認）
print("差分付きデータ（先頭 5 行）")
grid_counts.head()

# x_center, y_center をオルソ画像（GeoTIFF）の中心（UTM 座標）に合わせる
# 以前のコードですでに実行しているが、念のため画像全体の中心座標を取得しなおす
with rasterio.open(tif_path) as src: b = src.bounds x_center = (b.left + b.right) / 2
y_center = (b.bottom + b.top) / 2

#座標の確認（回転軸、回転後の座標サンプル、グリッドの基準点、グリッドサイズを表示）
print("x_center,y_center=", x_center, y_center)
print("x_rot,y_rot sample=", x_rot, y_rot)
print("x_min,y_min=", x_min, y_min)
print("grid_size_x, grid_size_y=", grid_size_x, grid_size_y)
```

（６）ヒートマップの作成

下記のコードは、一連の解析の最終成果物であるヒートマップを作成するための物です。単に色を塗るだけでなく、「樹列の傾きに合わせて文字を回転させる」、「外れ値を無視して色をきれいに割り振る」、「凡例（カラーバー）を付ける」といった可視化を行います。

```
# 9 段階ヒートマップ（回転グリッドの塗り）＋凡例＋diff 文字（約 100px）

# ライブラリの読み込み
import os
import cv2
import numpy as np
import rasterio
import matplotlib.pyplot as plt
from matplotlib.colors import BoundaryNorm    # matplotlib でデータを不連続（段階的）な色で表現する際に、値の範囲と色を対応付けるための機能

# ============================
# 設定
# ============================
# 保存先フォルダの作成（存在しなければ新しく作成する）
out_dir = "C:/Users/test/result"
os.makedirs(out_dir, exist_ok=True)

alpha_overlay = 0.40        # 透明度 0.3〜0.6 で調整（0.4 なら背景が 60%透ける）
cmap_name = "RdBu_r"   # 赤（多）〜白（適正）〜青（少）のカラーマップ
n_levels = 9                  # 色を何段階に分けるか

# 文字描画設定
DRAW_TEXT = True    # グリッドに数字を描写するか（True：する、False：しない）
text_target_h_px = 80    # 文字の大きさ（約 80 ピクセル）
text_min_abs = 0          # 0 なら全部表示、例えば 3 にすると |diff|>=3 の時のみ表示
text_color_fg = (255, 255, 255)   # 文字の色（白）
text_color_bg = (0, 0, 0)             # 文字の縁取り（黒）
```

```
# ==========================
# 入力チェック
# ==========================
# 必要なデータ（座標と差分）が揃っているかを確認
need_cols = {"grid_x", "grid_y", "diff"}
if not need_cols.issubset(set(grid_counts.columns)):
    raise ValueError(f"grid_counts に必要列 {need_cols} がありません。今の列
={grid_counts.columns.tolist()}")

# ==========================
# 背景画像読み込み
# ==========================
bg_img = cv2.imread(png_path)  # PNG のオルソ画像を読み込む
if bg_img is None:                    # PNG のオルソ画像が読めないとエラー表示
    raise FileNotFoundError(f"png_path が読めません: {png_path}")
bg_img = cv2.cvtColor(bg_img, cv2.COLOR_BGR2RGB)  # BGR から RGB に変換
h, w, _ = bg_img.shape      # bg_img の画像サイズを取得
overlay = bg_img.copy()  # 元画像を保護し、後で色を塗るためのコピーを作成

# ==========================
# グリッドの中央に回転した文字を描写
# ==========================
ct, st = np.cos(theta), np.sin(theta)  # 回転計算用の変数を用意

def rot_to_utm(xr, yr):      # 回転座標(xr, yr)を UTM 座標(x, y)に変換する
    x = x_center + (ct * xr - st * yr)    # 変換後の UTM 座標の X 成分を計算
    y = y_center + (st * xr + ct * yr)  # 変換後の UTM 座標の Y 成分を計算
    return x, y  # 計算された UTM 座標系上の点(x, y)を関数の結果として返す

# 背景色（塗りつぶし）付きの回転したテキストを重ねる
## 中心位置に angle_deg 回転した text を描く（縁取りあり）。
## target_h_px : 文字高さ（ピクセル）を目標に自動スケール。

def put_rotated_text_rgba(img_rgb, text, center_xy, angle_deg,
                          target_h_px=100, thickness=None,
                          fg=(255,255,255), bg=(0,0,0)):
```

```
    cx, cy = center_xy
    font = cv2.FONT_HERSHEY_SIMPLEX  # フォントを指定

    # scale=1.0 のときのテキストの高さから逆算してフォントの倍率（scale）を決定
    # 目標の高さ（target_h_px）を仮の高さ（th1）を割ることで倍率を算出
    (tw1, th1), base1 = cv2.getTextSize(text, font, 1.0, 2)
    font_scale = target_h_px / max(th1, 1)

    # 太さも自動で増やす（フォントスケールに合わせて線の太さも自動調整）
    if thickness is None:
        thickness = max(2, int(round(font_scale * 2)))

    # 計算したスケールと太さを使って、最終的なテキストの幅（tw）、高さ（th）、ベースライン（baseline：文字の下の余白）を取得
    (tw, th), baseline = cv2.getTextSize(text, font, font_scale, thickness)

    # RGBA キャンバスのサイズの決定（文字＋余白が収まる大きさ）
    pad = 10                        # 10 ピクセルの余白
    W = tw + pad * 2                # 文字の左右に 10 ピクセルの余白
    H = th + baseline + pad * 2     # 文字の上下に 10 ピクセルの余白

    # RGBA キャンバスの作成
    text_img = np.zeros((H, W, 4), dtype=np.uint8)
    org = (pad, pad + th)

    # 縁取り（黒、少し太い：thickness + 6、背景色：bg）
    cv2.putText(text_img, text, org, font, font_scale,
                (bg[0], bg[1], bg[2], 255),
                thickness + 6, cv2.LINE_AA)

    # 本体（白、本来の太さ：thickness、前景色：fg）
    cv2.putText(text_img, text, org, font, font_scale,
                (fg[0], fg[1], fg[2], 255),
                thickness, cv2.LINE_AA)
```

```
    # 回転
    center = (W // 2, H // 2)
    rot_mat = cv2.getRotationMatrix2D(center, angle_deg, 1.0)  # 回転の中心・角度
    rotated = cv2.warpAffine(                                  # キャンバスごと回転
        text_img, rot_mat, (W, H),
        flags=cv2.INTER_LINEAR,
        borderMode=cv2.BORDER_CONSTANT,
        borderValue=(0, 0, 0, 0)
    )

    # 回転したテキストの貼り付け位置の計算
    x1 = cx - W // 2
    y1 = cy - H // 2
    x2 = x1 + W
    y2 = y1 + H

    # 画像外なら描かない
    if x1 < 0 or y1 < 0 or x2 > img_rgb.shape[1] or y2 > img_rgb.shape[0]:
        return img_rgb

    # アルファブレンド（合成）：背景の上に前景をすかしながら重ねる
    roi = img_rgb[y1:y2, x1:x2].copy()
    alpha = rotated[:, :, 3] / 255.0    # 文字がある場所と無い場所のマップを作る
    for c in range(3):
        roi[:, :, c] = (1 - alpha) * roi[:, :, c] + alpha * rotated[:, :, c]
    img_rgb[y1:y2, x1:x2] = roi
    return img_rgb

# ==========================
# 9 段階カラーの境界を決める（固定 -50 〜 +50、超過は端色にクリップ）
# ==========================
vmin, vmax = -50.0, 50.0  # カラースケールの最小値を-50.0、最大値を 50.0 に設定

# 9 段階（10 個の境界）を設定
boundaries = np.linspace(vmin, vmax, n_levels + 1)  # -50〜50 を等間隔に区切り、10個の境界線を作成
```

```
cmap = plt.get_cmap(cmap_name, n_levels)                # 9 色に離散的な色を取得
norm = BoundaryNorm(boundaries, n_levels, clip=True)  # 超過を端色にする（-50 未満なら一番下の色、50 超過なら一番上の色に割り当て）

# 差分値（diff）を受け取って、9 段階 RGB(0-255)の数値を返す（-50〜50 の範囲）
def diff_to_rgb(diff):
    # BoundaryNorm(clip=True）で設定した norm が範囲外を自動処理
    idx = norm([diff])[0]      # 0..8（入力された diff を 0〜8 までの色番号に変換）
    rgba = cmap(idx)           # 0..1（赤・緑・青・透明度を 0.0〜1.0 で返す）
    return (int(rgba[0] * 255), int(rgba[1] * 255), int(rgba[2] * 255))  # 0.0〜1.0 を 0〜255 の整数に変換

# ==========================
# 回転グリッドを塗る + diff 文字
# ==========================
drawn = 0      # 描画したグリッドの数をカウントするための変数を 0 で初期化
skipped = 0    # スキップしたグリッドの数をカウントするための変数を 0 で初期化

# グリッドの座標と差分値の取得
with rasterio.open(tif_path) as src:
    for _, r in grid_counts.iterrows():
        gx, gy = int(r["grid_x"]), int(r["grid_y"])  # 列番号・行番号を整数型に変換
        diff = float(r["diff"])                      # 差分値を浮動小数点型に変換

        # ---- グリッドの 4 隅の座標を計算（rot 座標系） ----
        x1 = x_min + gx * grid_size_x
        x2 = x_min + (gx + 1) * grid_size_x
        y1 = y_min + gy * grid_size_y
        y2 = y_min + (gy + 1) * grid_size_y
        corners_rot = [(x1, y1), (x2, y1), (x2, y2), (x1, y2)]  # 四隅の座標リスト作成

        # ---- 座標変換（rot：回転座標 -> UTM 座標 -> pixel 座標） ----
        pts = []
        ok = True
        for xr, yr in corners_rot:
        x, y = rot_to_utm(xr, yr)      # 回転座標（rot）を UTM 座標に変換
```

```
# UTM 座標を pixel 座標に変換
row, col = src.index(x, y)                    # row=y, col=x
px, py = int(col), int(row)
pts.append([px, py])
# 計算されたピクセル座標が、画像の幅や高さの外に出ていないか確認
if not (0 <= px < w and 0 <= py < h):
    ok = False
# 四隅のうち一つでも画像の外にはみ出していたら描画しないで次に進む
if not ok:
    skipped += 1
    continue

# OpenCV で描画できるように座標リストを NumPy の整数配列に変換
pts = np.array(pts, dtype=np.int32)

# 色（9 段階）
color = diff_to_rgb(diff)

# 塗り
cv2.fillConvexPoly(overlay, pts, color)
drawn += 1

# ---- 中央に diff 表示 ----
if DRAW_TEXT and (abs(diff) >= text_min_abs):
    # グリッドの中心（rot 座標）を計算 -> UTM 座標 -> pixel 座標に変換
    cxr = x_min + (gx + 0.5) * grid_size_x
    cyr = y_min + (gy + 0.5) * grid_size_y
    x_c, y_c = rot_to_utm(cxr, cyr)
    row_c, col_c = src.index(x_c, y_c)
    cx, cy = int(col_c), int(row_c)

    if 0 <= cx < w and 0 <= cy < h:  # 座標が画像の範囲内にあるかチェック
        # 表示する文字の設定
        d = int(round(diff))  # 差分値を四捨五入して整数に変換
        text = f"{d:+d}"        # + を明示（例:+3,-2）
```

```
                # 差分値 diff>0 なら「赤字＋黒縁」、それ以外は「白字＋黒縁」
                if d > 0:
                    fg = (255, 0, 0)        # 赤
                    bg = (0, 0, 0)          # 黒縁
                else:
                    fg = (255, 255, 255)    # 白
                    bg = (0, 0, 0)          # 黒縁

                # 指定した位置・角度・色で差分値をオーバーレイに書き込む
                overlay[:] = put_rotated_text_rgba(
                    overlay, text, (cx, cy), angle_deg,
                    target_h_px=text_target_h_px,
                    fg=fg, bg=bg
                )

# ==========================
# 合成（背景 + オーバーレイ）
# ==========================
# overlay（色のついたヒートマップ等）と bg_img（元の背景画像）を混ぜる
# alpha_overlay：オーバーレイ側の不透明度（例：0.6 なら 60%）
# 1 - alpha_overlay：背景側の不透明度（例：0.4 なら 40%）
# 0：合成後に追加する輝度値（通常は 0）
vis_img = cv2.addWeighted(overlay, alpha_overlay, bg_img, 1 - alpha_overlay, 0)

# 処理状況と数値範囲の表示
print("描画できたセル数:", drawn, "/ スキップ:", skipped)
print("diff 範囲(表示用):", f"{vmin:.0f} 〜 {vmax:.0f}（超過は端色に丸め）")

# ==========================
# 画像保存（凡例なし）
# ==========================
# 保存パス（場所＋ファイル名）の設定
out_png = os.path.join(out_dir, "heatmap_9step.png")
# RGB から OpenCV 対応の BGR に変換して PNG 形式で保存
cv2.imwrite(out_png, cv2.cvtColor(vis_img, cv2.COLOR_RGB2BGR))
print("保存（凡例なし）:", out_png)
```

```
# ==========================
# 凡例付きで保存（matplotlib でカラーバーを付ける）
# ==========================
# 保存パス（場所＋ファイル名）の設定
out_png_legend = os.path.join(out_dir, "heatmap_9step_legend.png")
fig, ax = plt.subplots(figsize=(12, 10), dpi=200)  # 描画領域の準備
ax.imshow(vis_img)
ax.axis("off")              # グラフの縦軸・横軸（目盛りや枠線）を消す
ax.set_title("Flower Density Difference Heatmap (9-step)")  # 図の上にタイトル表示

# カラーバー作成
# カラーバー用の「空のカラーマップ」作成
sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
sm.set_array([])

# カラーバー（凡例）の描画
cbar = fig.colorbar(
    sm, ax=ax, fraction=0.045, pad=0.02,   # バーの大きさや画像との隙間の調整
    ticks=[-50, -25, 0, 25, 50]                  # バーに表示する目盛りの数値を指定
)
# カラーバーが何を表しているのか、説明テキストを付ける
cbar.set_label("diff = flower_count - target (per 2m×2m cell)")

# もし目盛り（ticks）が多すぎるなら間引き（必要なら下の行のコメントアウトを外す）
# cbar.set_ticks(boundaries[::2])

# タイトルやカラーバーが画面からはみ出さないよう、全体の配置を自動で最適化
plt.tight_layout()
fig.savefig(out_png_legend, bbox_inches="tight")  # 図をファイルに書き出す
plt.close(fig)      # 作成した図をメモリから削除（メモリの開放）

print("保存（凡例あり）:", out_png_legend)  # 完了メッセージの表示
```

- ✧ from matplotlib.colors import BoundaryNorm：連続的なカラーマップ（グラデーション）とは異なり、例えば「0〜10 は青、10〜20 は緑、20〜30 は赤」のように、境界（Boundary）を指定して色を区切りたい場合に使用します。
- ✧ cmap_name = "RdBu_r"：RdBu_r（**R**ed-**B**lu**e** の逆（**r**eversed））を使うと、花が足りないところが青、多いところが赤になり、直感的にわかりやすくなります。"bwr"だと青（少）〜白（適正）〜赤（高）となり、"RdBu_r"と似ていますが、色の飽和が早いため、データの中間値（白）からの変化を強調したい場合に適しています。
- ✧ figsize=(12, 10)：図の大きさを指定
- ✧ dpi=200：高精細（高解像度）な画像に設定

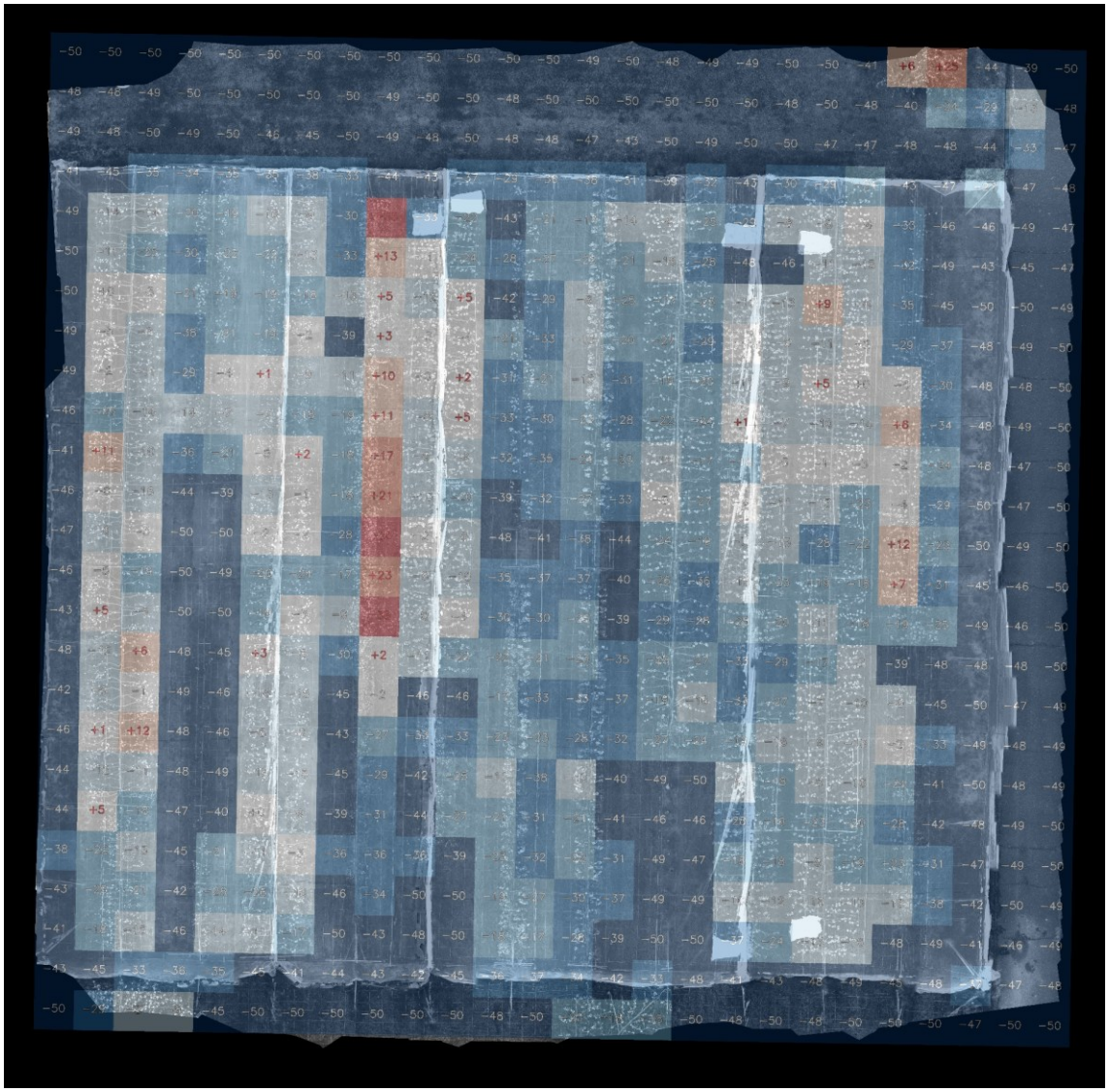


図２－１０　ヒートマップ（凡例なし）

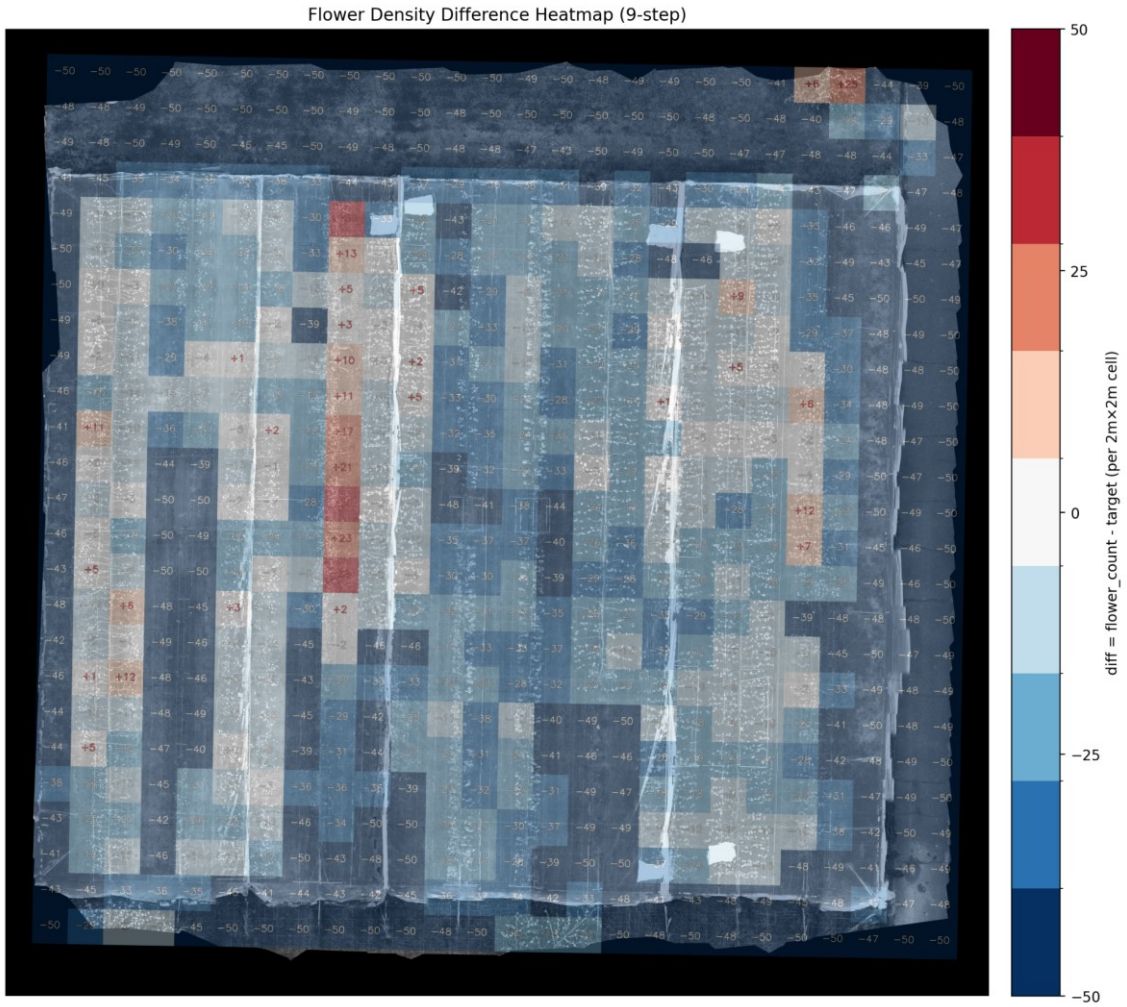


図２－１１　ヒートマップ（凡例あり）

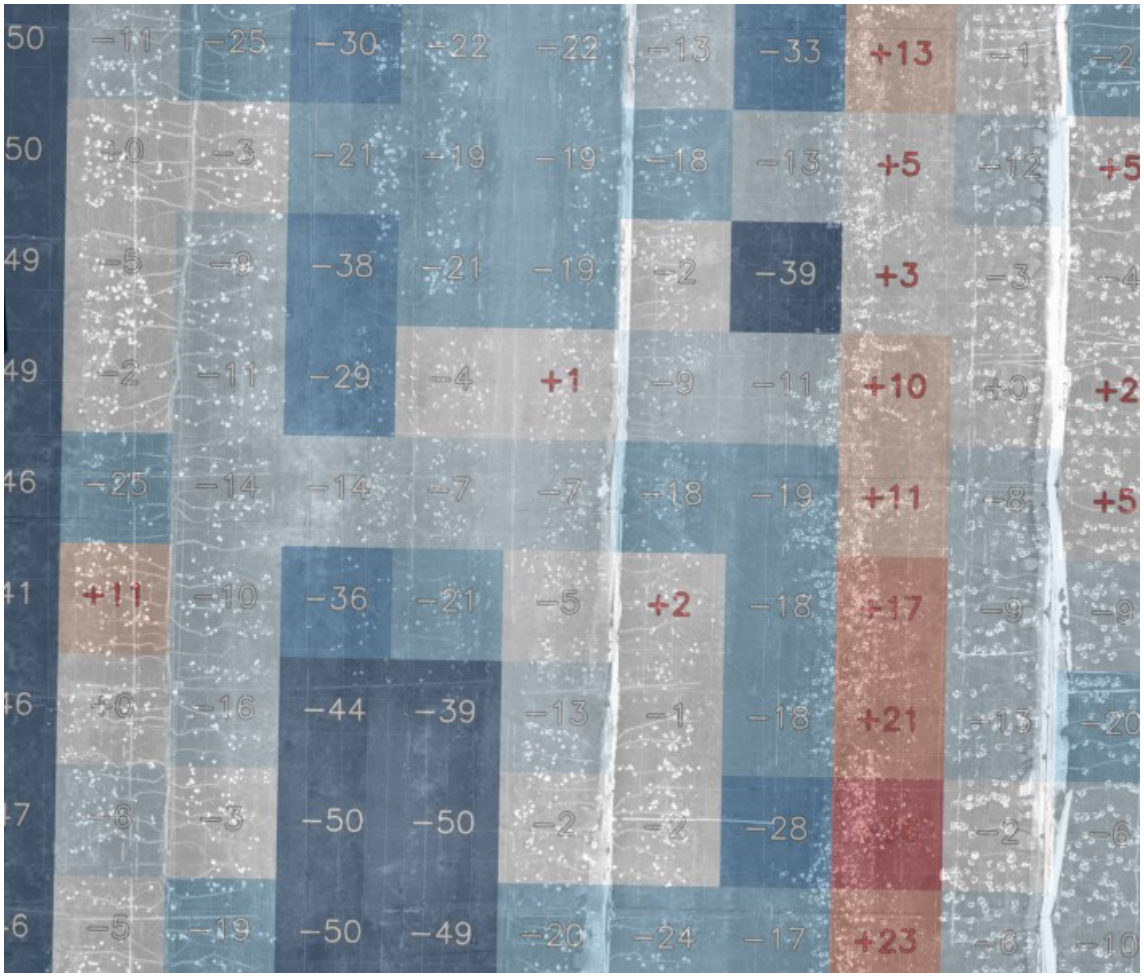


図２－１２　ヒートマップを拡大した様子

目標花そう数との差分を計算し、グリッドの色を変えるとともに、グリッド内に差分値を表示している

第3章　AIによる物体検出結果を営農管理ツールに連携

１．営農管理ツール「アグリノート」

ここではAIを使って作成したヒートマップを営農管理ツール「アグリノート」上に表示させる方法を紹介します。

（１）ヒートマップを「アグリノート」の立て看板にアップロード

アグリノートの「立て看板」機能を活用して、ヒートマップをアグリノート上にアップロードします。アグリノートの「立て看板」機は、図３－１のように地図（マップ）上の任意の位置に様々な種類のアイコンを配置し、メモ、写真、ファイルを共有できる機能です。

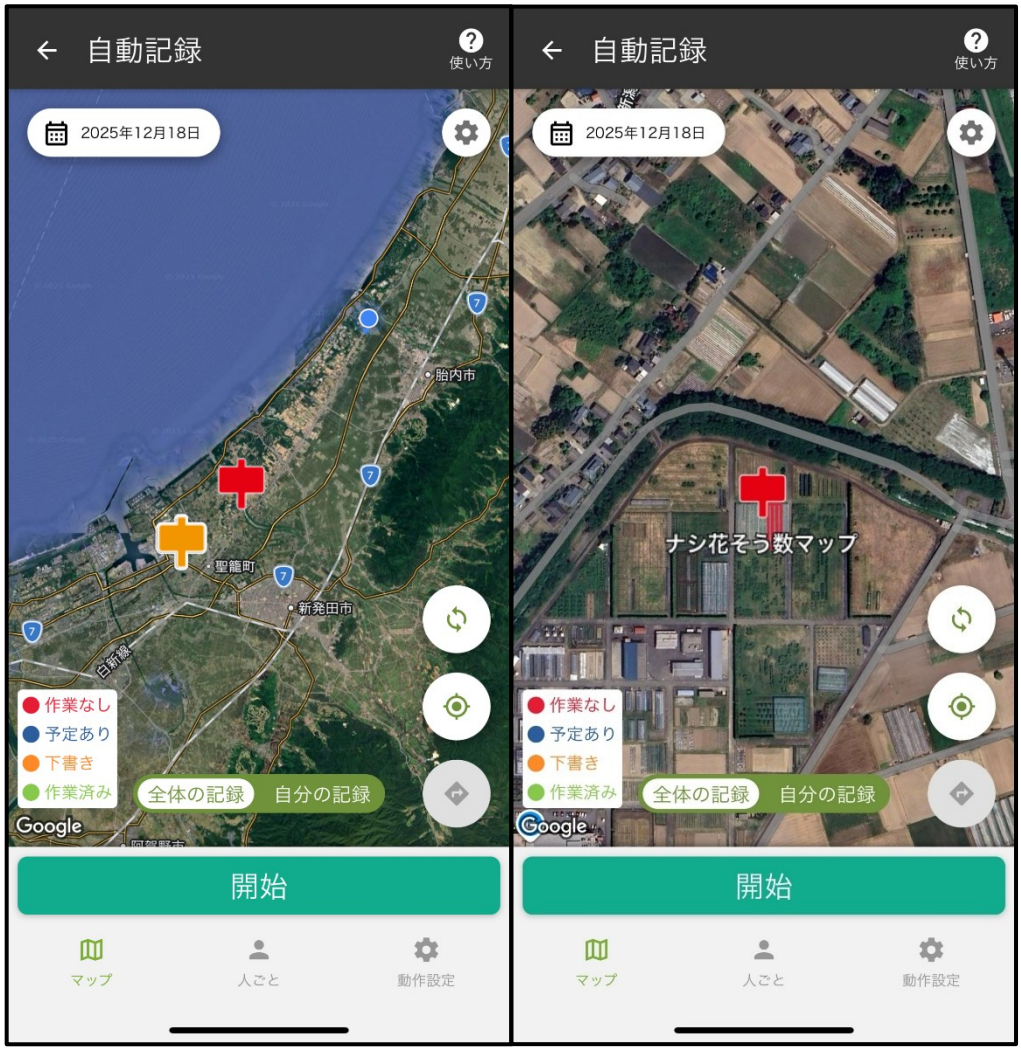


図３－１　アグリノート上に表示された立て看板

まずはパソコンでアグリノートを起動します。アグリノートの［マップ画面］の左下に［立て看板］のアイコンがあるので、クリックします（図３－２）。

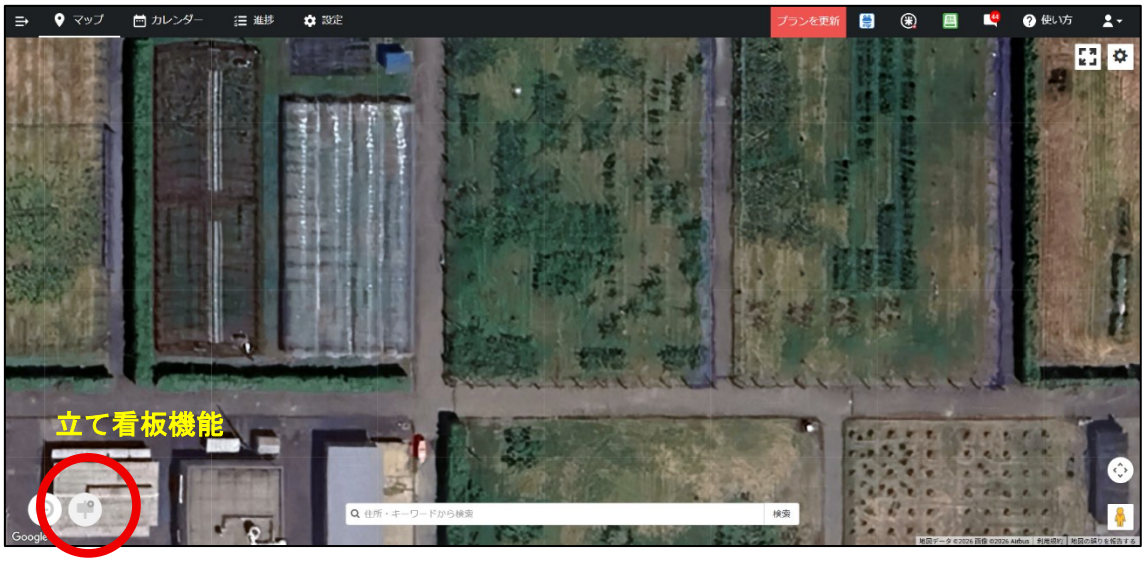


図３－２　マップ画面と立て看板機能のアイコン

画面の右側に立て看板の設定画面が表示されるので、［看板の名前］に看板の名前を入力し、［アイコンの選択］でアイコンを選択します。アイコンを未選択の場合は図３－３のような赤い看板が表示されます。立て看板の名前とアイコンが決定したら、［保存］をクリックします。

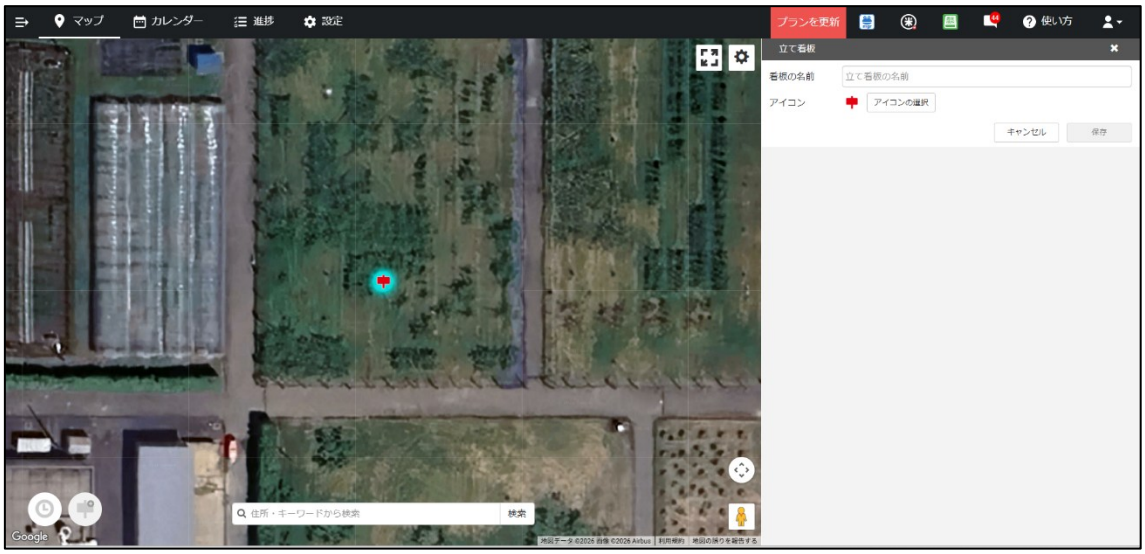


図３－３　立て看板の設定画面（画面右側）

図３－４　立て看板の設定画面（拡大）

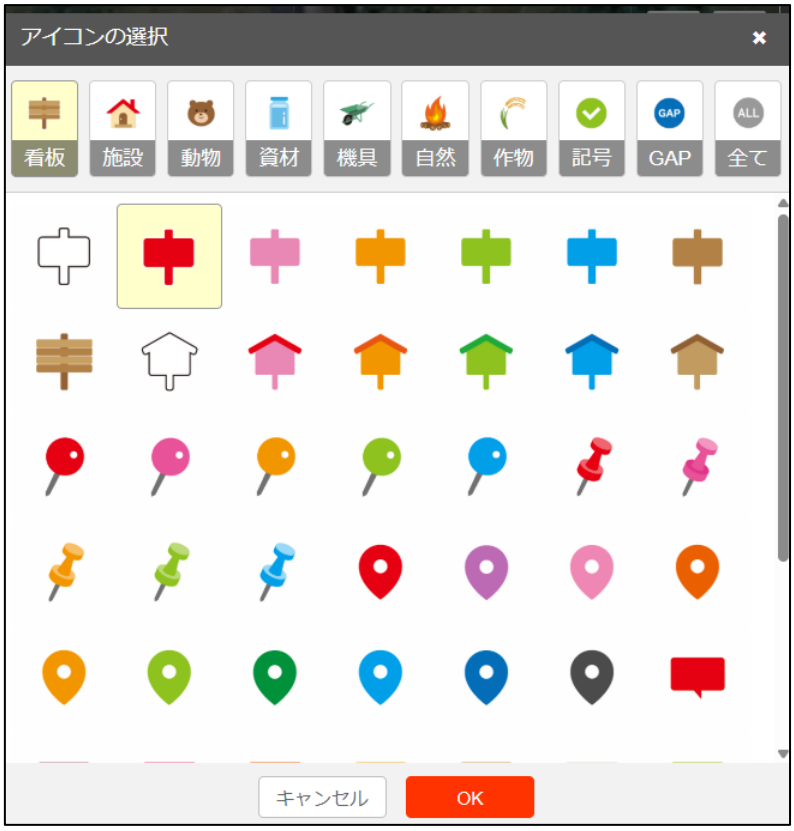


図３－５　アイコンの選択画面

立て看板の［メモ］画面が表示されます（図３－６）。立て看板に簡単なメモを書いたり、写真やファイルを添付したりすることができます。ここでは、作成したヒートマップを選択して［開く］をクリックします（図３－７）。

図３－６　立て看板の［メモ］画面

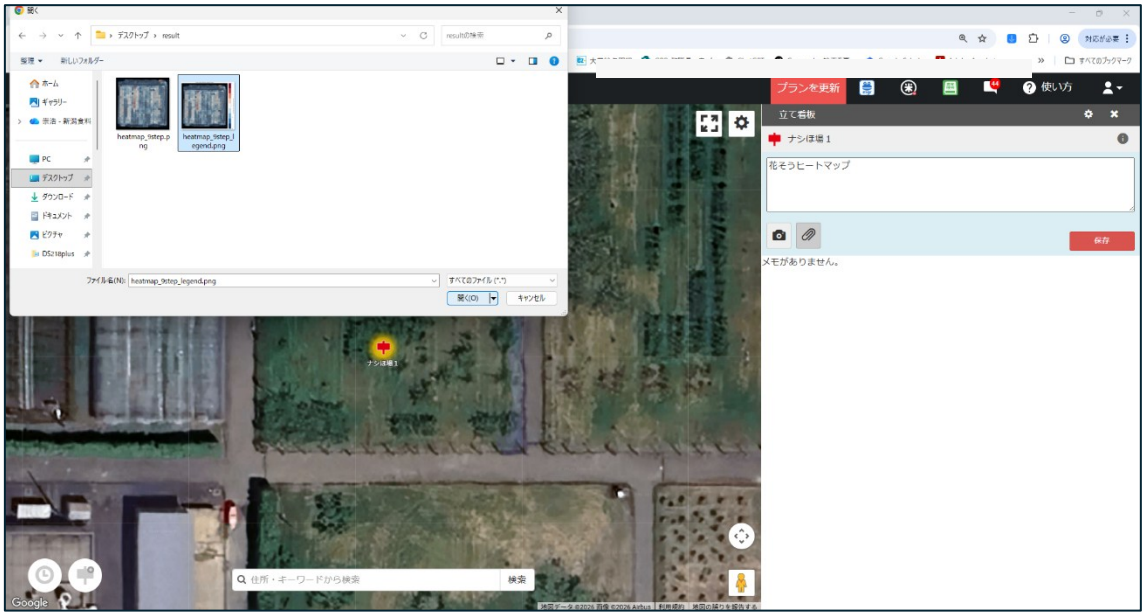


図３－７　［添付ファイル］の選択画面

ヒートマップをアップロードできたら、保存をクリックすると立て看板にメモや添付ファイルが登録されます（図３－８、図３－９）。

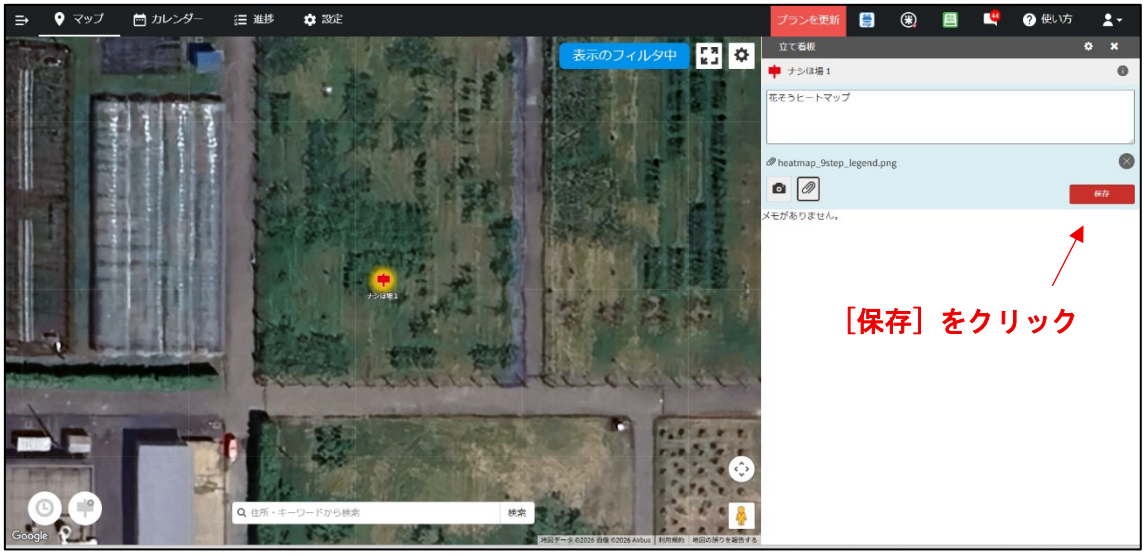


図３－８　添付ファイルをアップロードした画面

図３－９　立て看板に［メモ］と［添付ファイル］が登録された

立て看板に添付されたヒートマップをクリックすると、図３－１０のように表示されます。これでヒートマップを「アグリノート」の立て看板にアップロードする作業は完了です。

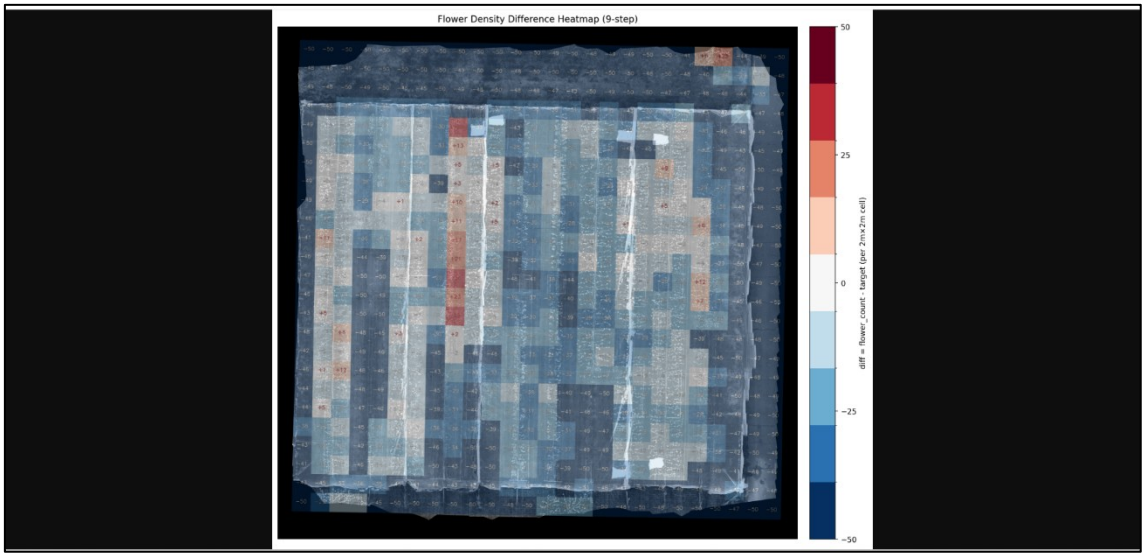


図３－１０　パソコン画面上で表示されたヒートマップ

（２）スマートフォンの「アグリノート」アプリでヒートマップを閲覧

スマートフォンの「アグリノート」アプリを使用して、立て看板に登録されたヒートマップを閲覧する方法を解説します。

まず、スマートフォンの「アグリノート」アプリを起動し、［マップ］画面を開きます。

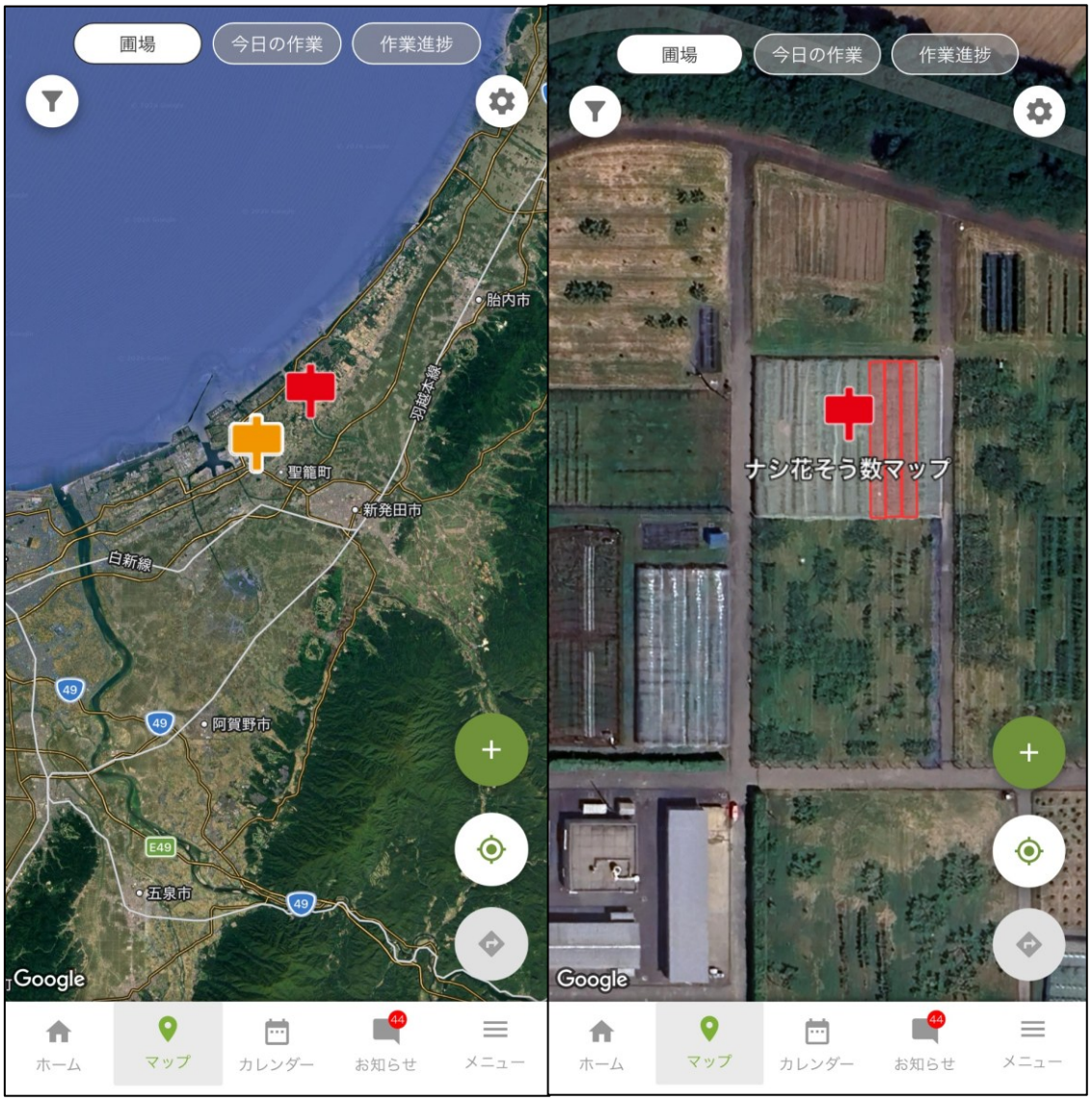


図３－１１　アグリノートの［マップ］画面

左：広域のマップ、右：マップを拡大した様子

立て看板をタップすると、メモや添付ファイル（赤枠部分）が表示されます。添付ファイルをタップすると、添付ファイルがダウンロードされます。

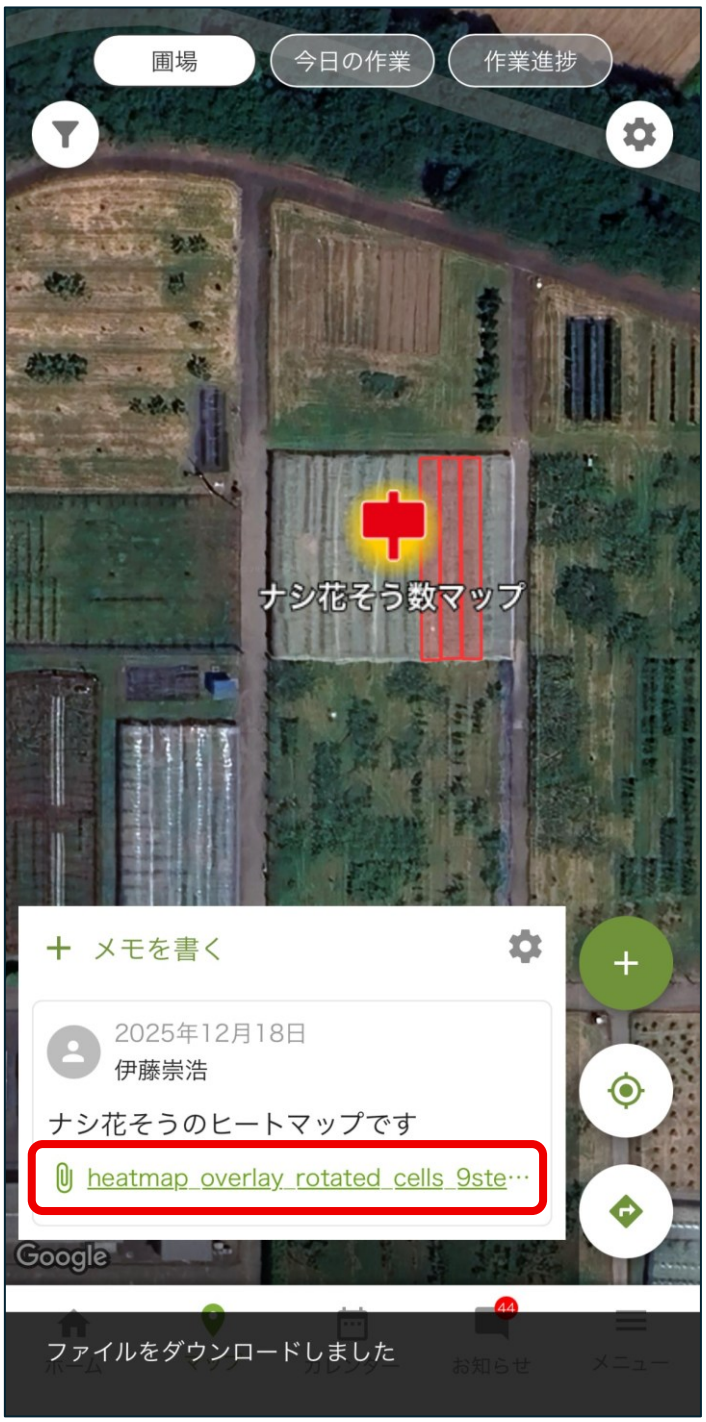


図３－１２　立て看板のメモと添付ファイル

iPhone の場合は、ダウンロードした添付ファイルが［ファイル］に保存されます（図３－１３）。［ファイル］を開き、［ブラウズ］→［この iPhone 内］→［アグリノート］→［downloads］を選択すると、ダウンロードしたヒートマップのファイルがあります（図３－１４）。

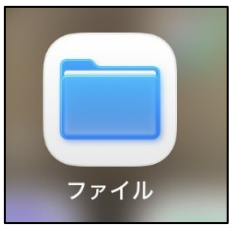


図３－１３　iPhone の［ファイル］のアイコン

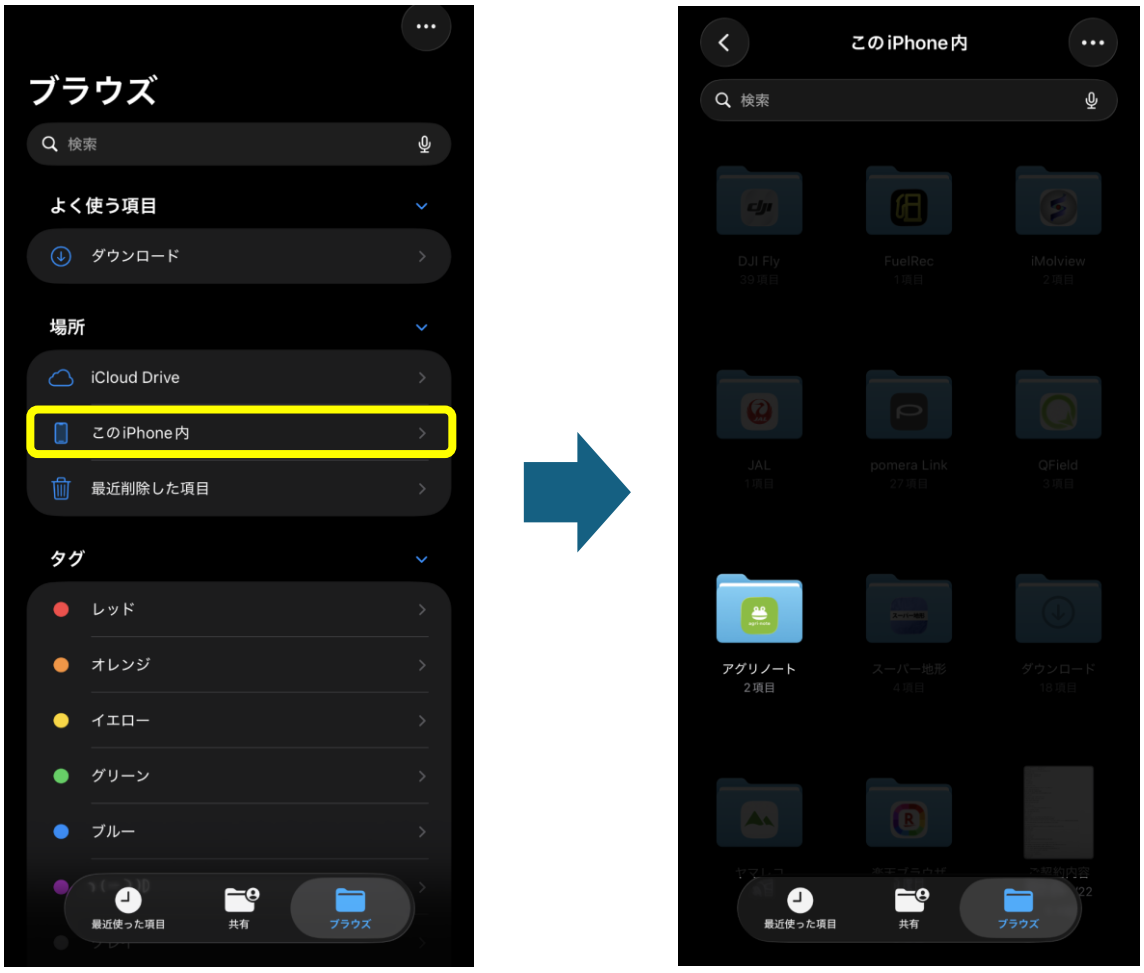


図３－１４　ダウンロードしたヒートマップの場所（［この iPhone 内］→［アグリノート］）

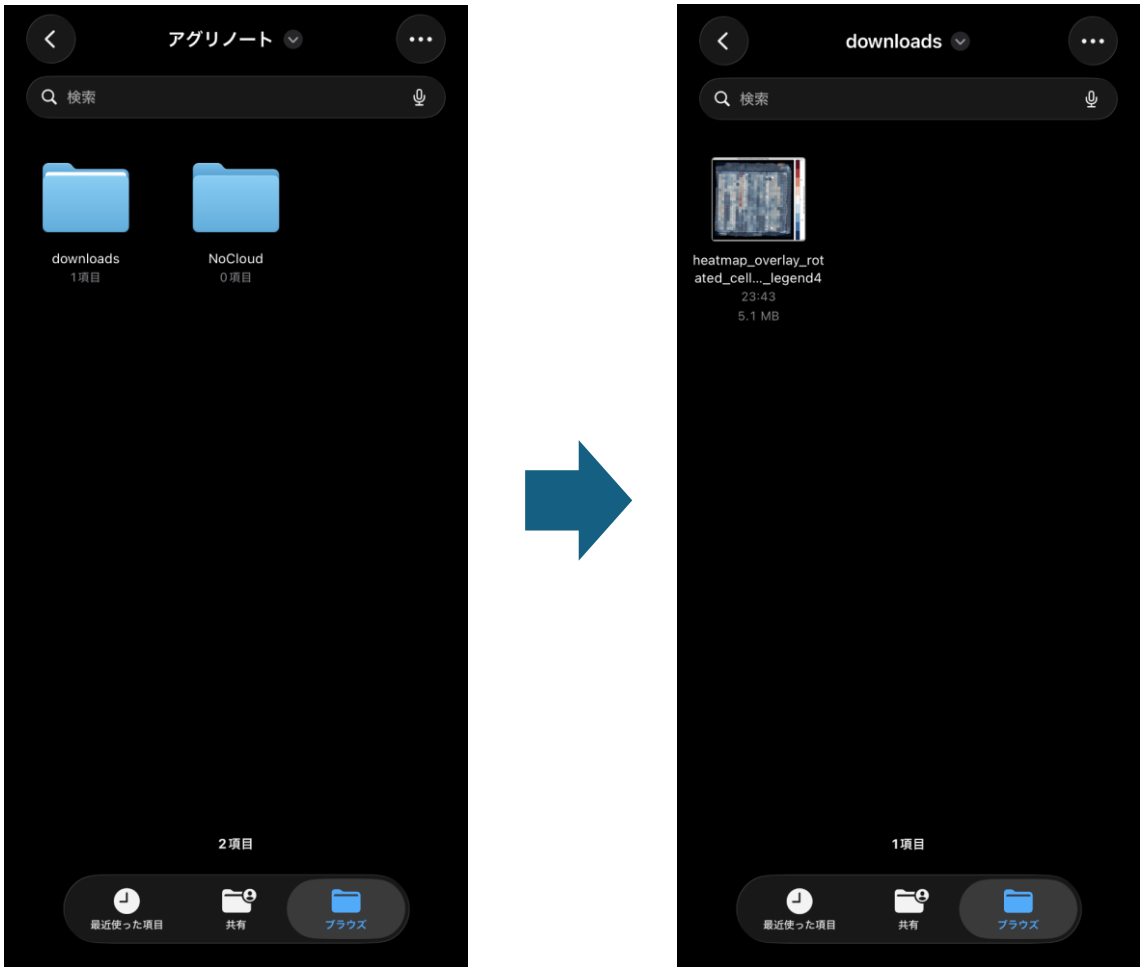


図３－１５　ダウンロードしたヒートマップの場所（[downloads]）

［downloads］内にあるヒートマップのファイルをタップすると、画像が表示されます（図３－１６）。

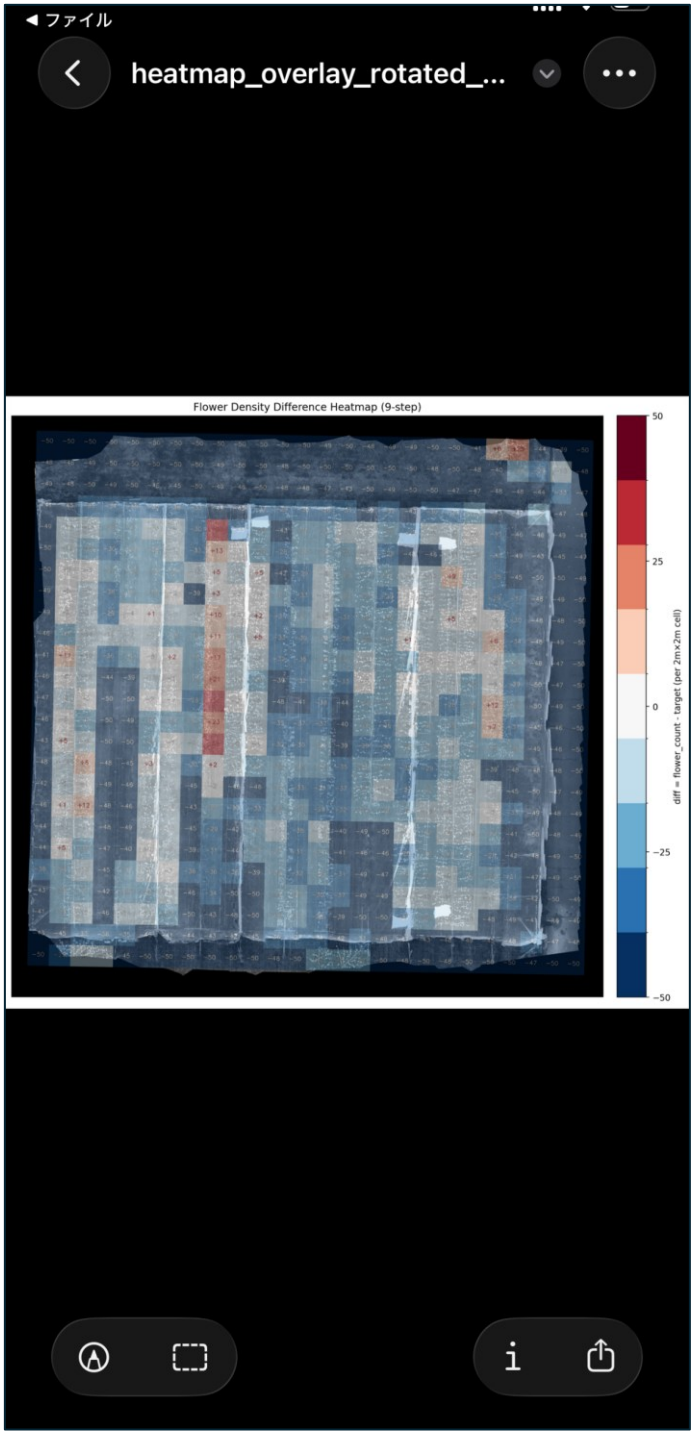


図３－１６　iPhone 上に表示されたヒートマップ

その他の情報

本資料は生研支援センターの「令和４年度補正予算及び令和５年度当初予算　戦略的スマート農業技術の開発・改良」
により実施した研究成果に基づき編集しています。

〈研究課題名〉　ドローン画像を利用した果樹の開花着果状況、病害発生状況の解析に基づく効率的栽培管理技術開発

〈研究期間〉　令和５年度～令和７年度

〈研究担当機関〉
新潟食料農業大学
新潟県農業総合研究所園芸研究センター
富山県農林水産総合技術センター園芸研究所果樹研究センター
株式会社プログレス
聖籠フルーツビレッジ

〈問い合わせ先〉
コンソーシアムに関するお問い合わせ
新潟食料農業大学　総務部　経理・研究支援課
新潟県胎内市平根台 2416
TEL　0254-28-9828
Mail　keiri@nafu.ac.jp

〈発行〉令和８年３月 10 日

〈編集・発行〉新潟食料農業大学