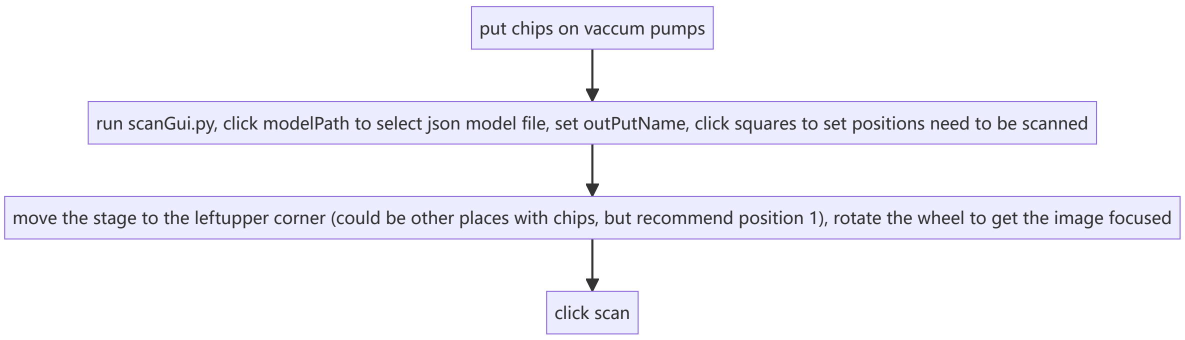
Documentation of AutoFlakeSearching

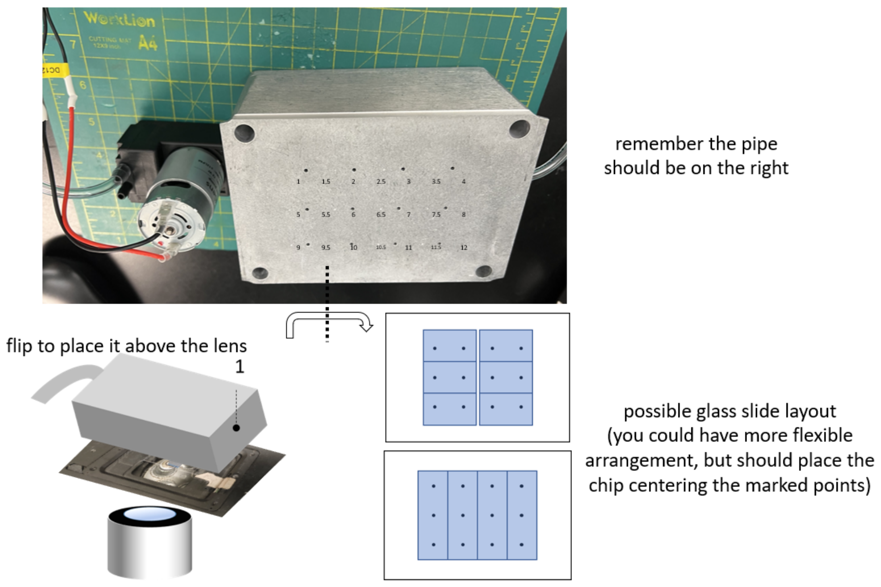
Documentation of AutoFlakeSearching

Short Guide

Short Scan Guide


put chips on vaccum pumps
run scanGui.py, click modelPath to select json model file, set outPutName, click squares to set positions need to be scanned
move the stage to the leftupper corner (could be other places with chips, but recommend position 1), rotate the wheel to get the image focused
click scan

Place Chips

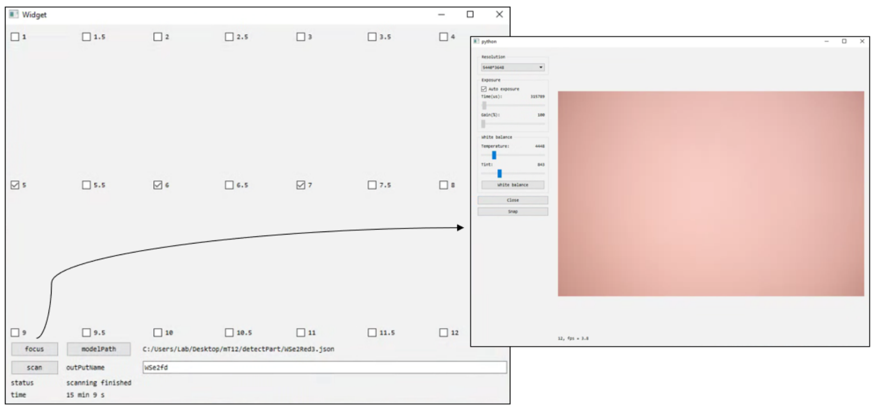

remember the pipe should be on the right
flip to place it above the lens
1
possible glass slide layout (you could have more flexible arrangement, but should place the chip centering the marked points)

Remember before placing the vacuum box on the microscope, open switch of vacuum pump first.

Note: no matter how many glass slides you may want to scan, you should use bare glass slides to seal the holes of the box to make the vacuum pump stick the glass slides.

Run Gui File

Run `scanGui.py`.



```
modelPath   C:/Users/Lab/Desktop/mT12/detectPart/WSe2Red3.json
outPutName  WSe2fd
status      scanning finished
time        15 min 9 s
```

1. Click `modelPath` to load the model `.json` file. Set `outPutName`.
2. Click the squares to select the areas to be scanned.
3. Click `focus`, you will see a newly created window showing CCD images. You should move it to a place with features and adjust the microscope focus wheel to get focused.
4. Click scan to start running. Notice `status` and `time`, it will show the status after it is finished. During scanning, you will see 3 opencv image windows, corresponding to the focus, boundary-finding, and chip-scanning process. Another parallel program is running for the detection of scanned images.

Note: you should check the **flatfield image** (just use light with constant intensity to illuminate a featureless flat substrate) in the suitable position. Details could be referred from later parts.

Short Train Guide

Run Notebook

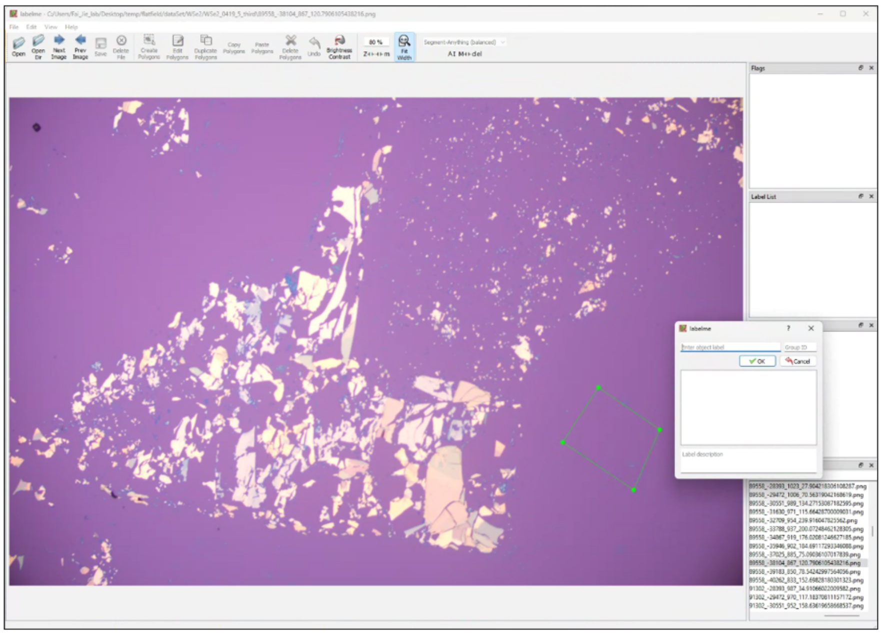
Run `train.ipynb`. For extra questions of `ipynb` file, you could look it up in **Package Requirement** Part.

Flatfield applying to folder

If you want to make the images in folders flatfielded, apply it.

In current version, we have done this process in scanning process, so you can just skip it. Please check carefully about the data you collected

Annotation Process



```
89558_-28393_1023_27.904218306108287.png
89558_-29472_1006_70.56319042168619.png
89558_-30551_989_134.27153087182595.png
89558_-31630_971_115.66428700009031.png
89558_-32709_954_239.916047825562.png
89558_-33788_937_200.07248462128305.png
89558_-34867_919_176.02081246627185.png
89558_-35946_902_184.69117293346088.png
89558_-37025_885_75.09036107017839.png
89558_-38104_867_120.7906105438216.png
89558_-39183_850_78.54242997564056.png
89558_-40262_833_152.69828180301323.png
91302_-28393_987_34.91066022009582.png
91302_-29472_970_117.18370811157172.png
91302_-30551_952_158.63619658668537.png
```

1. Click `Labelme.exe`, and select the folder to be annotated.
2. Click `Create Polygons` on the upper bar. Draw polygons to mark the flakes and text your labels on them.
3. Save the `.json` file in the same folder (`ctrl+s`).

You could also install an anaconda version of `labelMe` and use `conda` commands. For details see official documentation.

Json Information to Mask Generation

After annotation, it could only generate `.json` file, which records the edge and image information. To utilize that data in our codes, we need to transform it to mask images, where the flake area have different pixel values than other areas.

1. Define `jsonDir` and `saveDir` (normally `None`, so you could just skip setting it).
2. Run `maskDir = jmk.json2MaskDir(jsonDir=jsonDir, saveDir=saveDir)` cell, get `maskDir` defined.

Note: in some cases, like clarifying TMD layers, the categories could be determined by number of TMD layers (like monolayer or bilayer). However, in graphite or hBN, whose categories are mainly determined by exact numerical thickness values (like 3um or something else). In that case, there maybe issues about how to divide the clustering (or in other words, how many categories should we divide them since their thicknesses are so continuous). I suggest using thicks in `annotation` process:

- When using `labelme`, you should have determined the thickness selection method (like you decided to divide them into categories like around 5um, 10um, 15um ...), then you text the thickness label in `labelme` (like 5, 10, 15 ...).

Normally, the labels in `.json` file are useless since often we just cluster the pixels from the ground up. Currently, there is a file 'label.txt' in `saveDir`, counting all the labels you have used in annotation. I think in future we could use them for:

1. the number of labels to determine the number of clusters
2. the names of clusters, making them look more friendly

Get Contrast from Data Points

Get the pixel values of the areas you have selected and compute the contrast.

1. Set `imgDir` and `maskDir` (could be the `jsonDir` and `maskDir` previously since these two are folders having images and masks)
2. Run `dataContrast = gct.get_contrasts_from_dir(image_directory=imgDir, mask_directory=maskDir)` cell.

Note: due to the intrinsic algorithm, if you find flake images that contain the edges of substrates, you would better not include that in your dataset, since the background colors may have errors.

Correlation Heatmap Plot & Automatic Data Points Crop

Show the heatmap of contrast data points. The orange dashed lines will show the autoCropped data points area.

Just run `hmac.heatMapAutoCropPlot(data=dataContrast, autoCropRatio=0.5)` (set the ratio as you want) cell.

- If you want to use fewer channels, not the full RGB, add parameter `used_channels='BR'` or something else.
- If you are satisfied with the `autoCrop` results, you could use `dataAutoCrop, boundMin, boundMax = hmac.heatMapAutoCropPlot(data=dataContrast, ...)` to save the `autoCrop` data for further usage.

Gaussian k-means Clustering for Contrast Data

Just run the cell.

- You should choose the data you want to train (set parameter `data=`, like `dataContrast` or `dataAutoCrop`).
- You should choose the number of categories you want to divide (set parameter `num_components=`).
- If you think there are noises in clustering, you could throw out them. Just set `num_additional_noise_comp` and change the `num_components` value adding `num_additional_noise_comp`
- Get return value of `all_means_gauss, all_covariances_gauss, all_weights_gauss, sampled_data, predicted_labels`

Draw Gaussian Fit Data

Draw heatmaps, with ellipses of confidence and histograms with Gaussian fit curves.

Just run `gpt.plot_gaussians(data=dataAutoCrop, predicted_labels=predicted_labels, gauss_means=all_means_gauss,gauss_weights=all_weights_gauss, gauss_covariances=all_covariances_gauss,);` cell. Note the data you want to draw.

Export Json File

Generate `.json` file for Gaussian clusters. This is the final model data.

Just run `cjn.constrast2Json(gaussMean=all_means_gauss, gaussCov=all_covariances_gauss, saveName='WSe2Red3')` cell.

- Set parameter `saveDir=` to the place you want to store the model file or just default `None` to save it in the same folder of `train.ipynb` file.

Short Read Guide

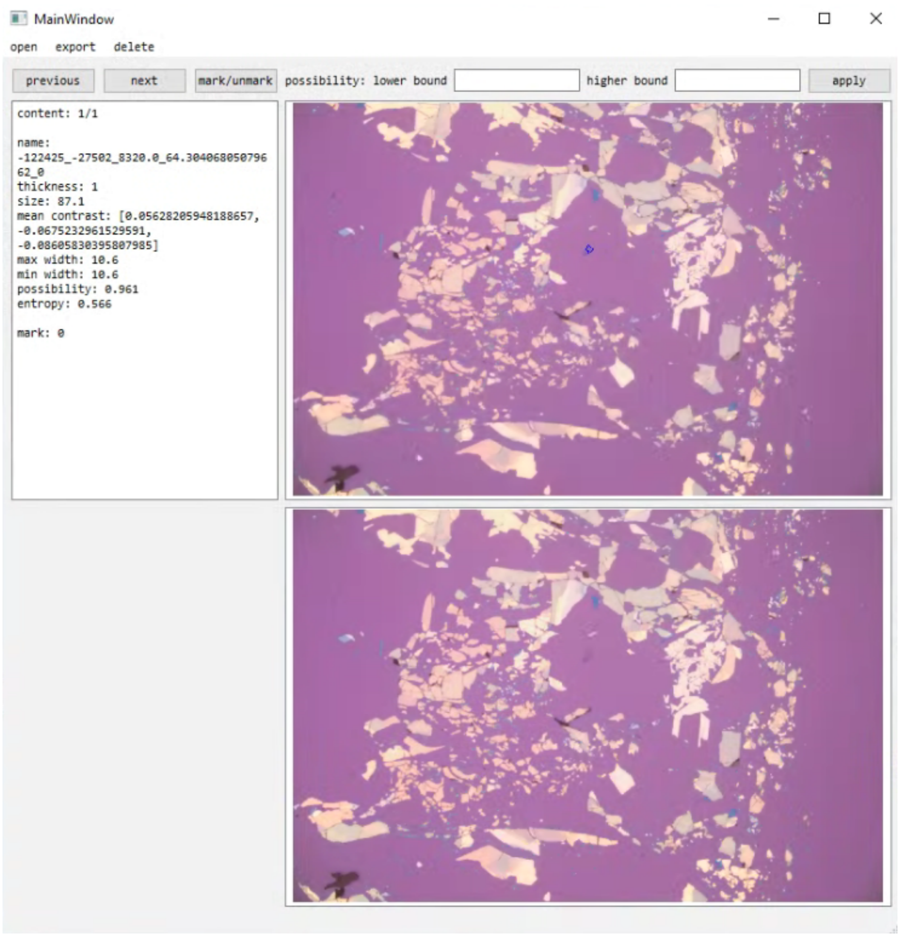
Our algorithm has the initial parallel process with scanning and detecting simultaneously. After detecting, we would generate folders including:

- `outPutName` (you set in **Short scan guide**) + `_#` (`#` is the number of coordinates in the box, see **Short scan guide, Place Chips**) folders.
- `outPutName_#Detect` folders.

The second kind of folder includes images we select that possibly have flakes and `json` files of the flakes information.

We should check them finally manually.

Run GUI File



```
content: 1/1

name:
-122425_-27502_8320.0_64.304068050796
62_0
thickness: 1
size: 87.1
mean contrast: [0.05628205948188657,
-0.0675232961529591,
-0.08605830395807985]
max width: 10.6
min width: 10.6
possibility: 0.961
entropy: 0.566

mark: 0
```

1. Run file (`readPart\ui.py`)
2. Click `open` for corresponding folders. You should follow the instructions, firstly open the scanned image folder, then the folder with detected information.
3. Then point `previous` and `next` to go through all the images detected in the folder. In the right panel showing images, the upper part is images with blue lines circling the potential flakes and the lower part is images that are original after scanning.
4. Click `mark/unmark` to decide whether you select this image or not. The information of `mark` is in the upper left information panel (`mark=0` or `mark=1`). The unit of width and size is μm.
5. You could delete current images if you find the current image is not the correct identification of flakes. Click `delete`, then select `delete current`. Or you could just `mark` the images you would like to accept, then click `delete` and `delete unmarked` in just one step finally.
6. After looking through all the images. You could select `export`:
   - Click `export current train data` to export current display image train data to a folder.
   - Click `export marked train data` to export all marked images with mask information to source image and mask data folders. It enables you to accumulate data for **train** datasets.
   - Click `export current world map` to export current display image world map to a folder.
   - Click `export marked world map` to export all marked images with world map and zoom-in image to the data folder. It enables you to save the flake images for searching under microscope.

7. **Optional**: Based on the criteria of shape (see arXiv:2306.14845), we could use machine learning to estimate the possibility of shape characteristics with the correctness of flake identification. The upper row enables you could set `lower bound` and `higher bound` to select the flakes satisfying the conditions. However, in practice, I think it may not be so effective when image number gets too big. You could just skip it.

Keyboard Shortcuts

Commands	Keyboard shortcuts
`previous`	`a`
`next`	`d`
`mark/unmark`	`s`
`delete current`	`x`
`delete unmarked`	`Ctrl` + `x`

Manual Detection

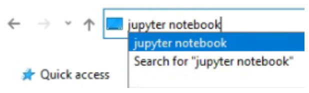
if there are issues that the detection works not so well. You could detect manually using `detect.ipynb`.

- Run `folder detection` cell, set `path` as the folder you want to detect. You could change `thres`, the threshold of Mahalanobis distance. When it get larger, meaning there maybe more false images get detected but the tolerance it bigger. When it get smaller, there maybe less false images but the tolerance is tight. So you need to test the balance of such trade-off.
- If you find cut pixel width of image overlap areas is not numerically suitable. You could practice multiple times in `world map generation` for better values.

Package Requirement

Run ipynb file

To run `.ipynb` file, I recommend downloading `vscode`. However, there are many other methods:



- Text `jupyter notebook` or `jupyter lab` and `enter` to open jupyter notebook/lab.
- Install `spyder notebook` to run `.ipynb` file in `spyder`. You should run the command `conda install spyder-notebook=0.4.1 -c conda-forge`. There is official documentation here. This may not be so stable.

Package Version Requirement

Installing packages has two ways:

- Run `conda install` + name of packages or something else. You could search the package with different versions in Package repository for Anaconda. Recommended
- Run `pip install package_name==version_number`. Brute force.

I recommend using earlier packages since they are more stable. My test environment is `python 3.9/3.10`.

Package	Version
`scikit-image`	>=0.19.3
`scikit-learn`	>=0.24.2
`opencv-python`	My current version is 4.5.1.48, but need to install it with `pip`
`pyvisa`	My current version is 1.13.0 (for scanning control)

Also, check if you have installed `joblib` and `watchdog`. You can check all your installed packages in `Anaconda Prompt` using `conda list` command.

Possible commands:

```
conda install -c fastchan scikit-image
conda install -c cctbx202112 scikit-learn
pip install opencv-python==4.5.1.48
```

Code Guide

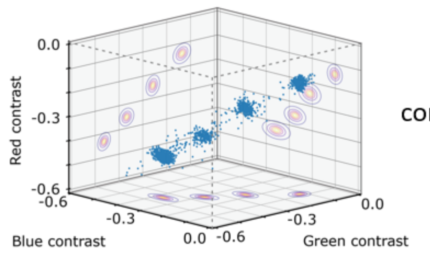
The structure of all the files:

```
|   detect.ipynb
|   find.py
|   RunFile.py
|   scanGui.py
|   scanUi.py
|   train.ipynb
|
├─detectPart
|       detector.py
|       findFlake.py
|       flakeClass.py
|       flakeOutput.py
|       flakeVisualize.py
|       imgCheck.py
|       shapeModel.joblib
|       model json files ...
|
├─readPart
|       flakeLoad.py
|       form.py
|       ui.py
|       worldMap.py
|
├─scanPart
|       amcam.dll
|       amcam.py
|       AmScope.py
|       chipScan.py
|       focusUi.py
|       microscope.py
|       priorMover.py
|       sample.py
|       sampleInit.py
|       third.png # flatfield img, could be replaced
|
└─trainPart
        contrast2Json.py
        flatfield.py
        gaussianKMeans.py
        getContrast.py
        guassianPlot.py
        heatMapAutoCrop.py
        json2Mask.py
```

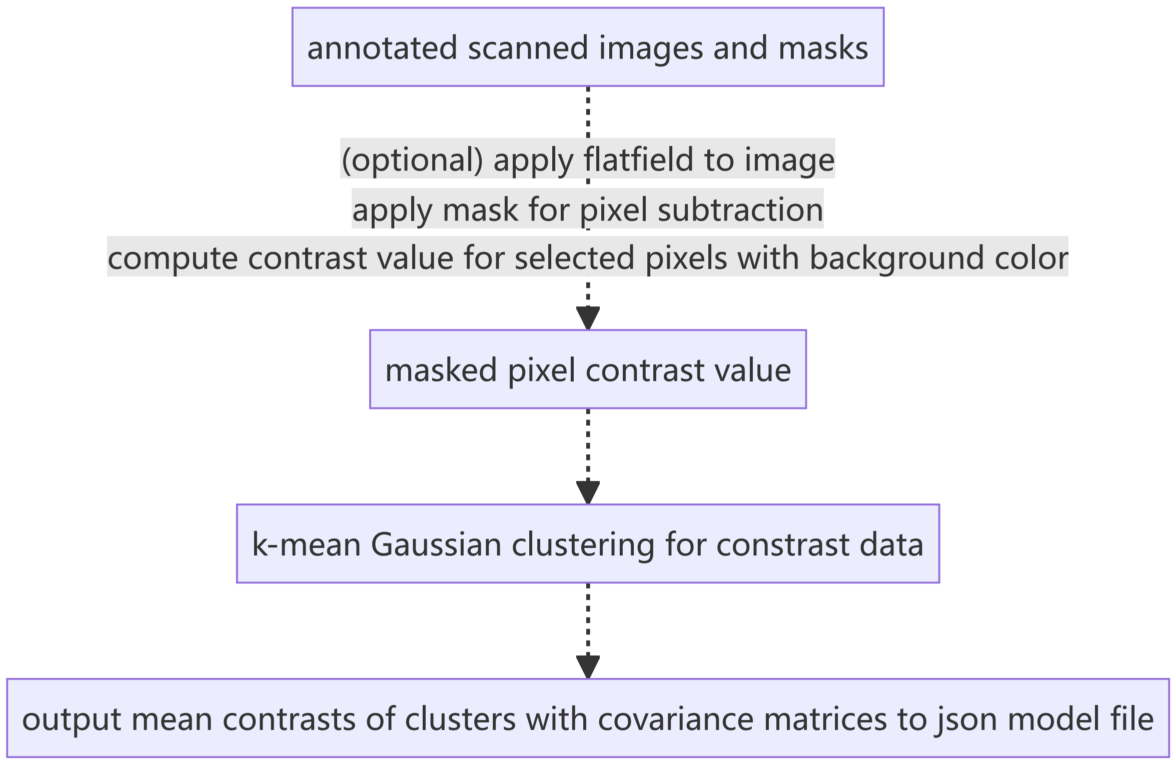
Basic Algorithms

Refer: [arXiv: 2306.14845]

Train

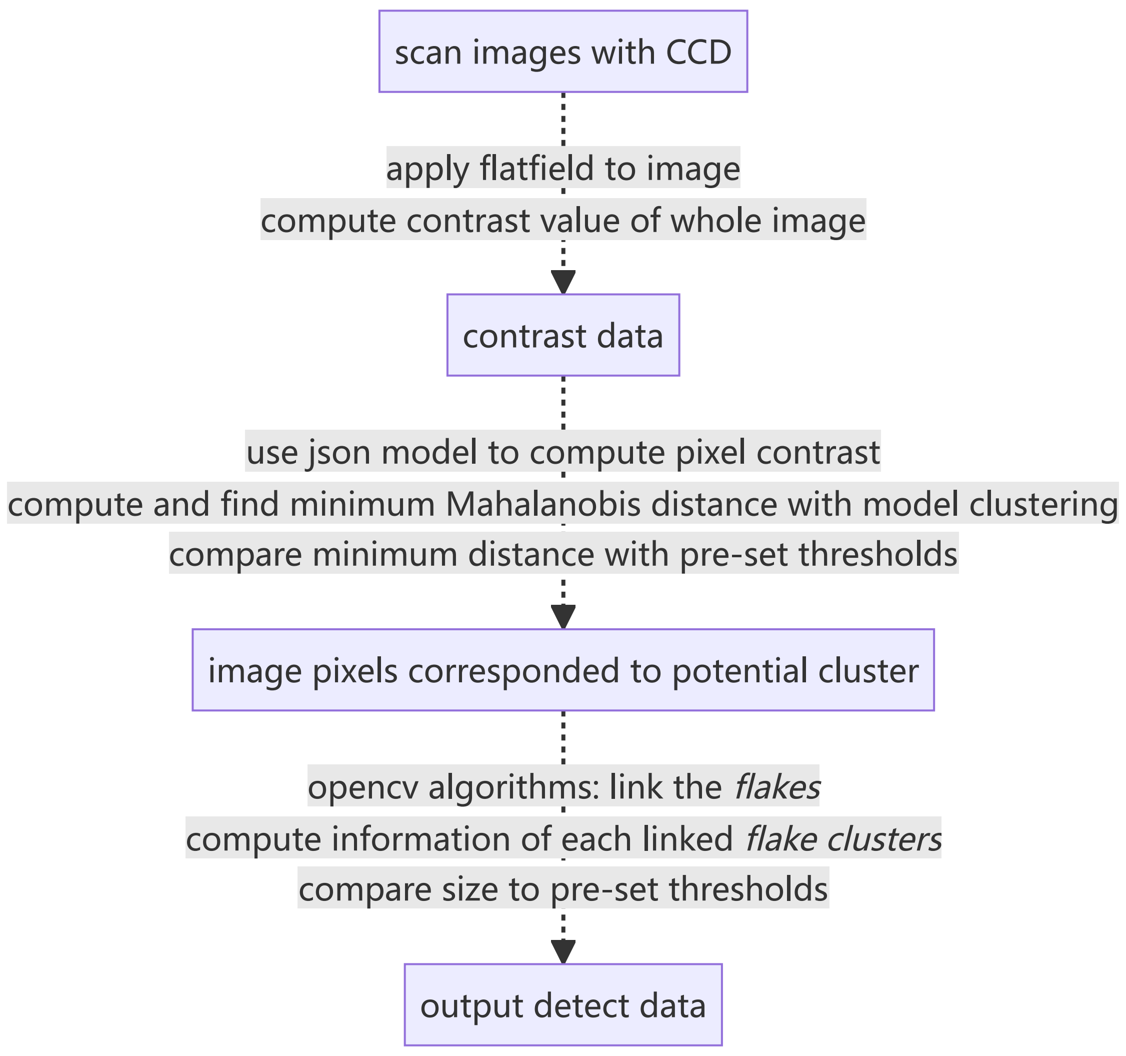


contract of different thickness flakes could be separated in color space


annotated scanned images and masks
(optional) apply flatfield to image
apply mask for pixel subtraction
compute contrast value for selected pixels with background color
masked pixel contrast value
k-mean Gaussian clustering for constrast data
output mean contrasts of clusters with covariance matrices to json model file

Ps: $\text{pixel contrast} = \frac{\text{pixel BGR}}{\text{background BGR}}$.

Detect



scan images with CCD
apply flatfield to image
compute contrast value of whole image
contrast data
use json model to compute pixel contrast
compute and find minimum Mahalanobis distance with model clustering
compare minimum distance with pre-set thresholds
image pixels corresponded to potential cluster
opencv algorithms: link the *flakes*
compute information of each linked *flake clusters*
compare size to pre-set thresholds
output detect data

Mahalanobis distance: for arbitrary vector $\vec{x}$ and cluster mean vector $\vec{\mu}$, with covariance matrix S

$$d_M(\vec{x}) = \sqrt{(\vec{x}-\vec{\mu})^T \cdot S \cdot (\vec{x}-\vec{\mu})}$$

Details

Scan Part

priorMover

Documentation: ProScan III Controller

`Class Prior`:

- `init`:
  - `Parameters`:
    - `self`
    - `addr`: link controller address.
    - `shortSleepTime`: `float = 0.03`, set sleep interval (short) for programs.
    - `longSleepTime`: `float = 0.05`, set sleep interval (long) for programs.
    - `debug`: `bool = False`, set to `True` for debug process.

    - Create a mover instance, with pre-setting velocity and acceleration, length units and moving directions.
- `__del__`:
    - Delete the instance
- `setCmd`:
    - `Parameters`:
        - `self`
        - `command`: `str`, please refer to controller documentation for cmd details.
    - Send controller command, suitable for all kinds of cmds.
- `setZVeloAcce`
    - `Parameters`:
        - `self`
        - `velo`: `int=None`, set velocity (1-100).
        - `acce`: `int=None`, set acceleration (1-100).
    - Set or view velocity and acceleration for Z axis.
- `set_units`:
    - `Parameters`:
        - `self`
        - `zUnit`: `int=50`, set z-axis length units.
        - `xyUnit`: `int=25`, set xy-axis length units.
    - Set units of moving steps (x, y, z).
- `set_direction`:
    - `Parameters`:
        - `self`
        - `xDir`: `int=-1`, set x-axis moving direction (-1,1).
        - `yDir`: `int=1`, set y-axis moving direction (-1,1).
        - `zDir`: `int=-1`, set z-axis moving direction (-1,1).
    - Set x, y, z directions.
- `setZero`:
    - `Parameters`:
        - `self`
    - Reset and move to the original points (0,0) in xy-panel.
- `getPos`:
    - `Parameters`:
        - `self`
        - `category`: `str`. Category of input coordinate ('X', 'Y', 'Z', ' ').
    - `Return`:
        - `list` of `int`, position list.

    - Get current position, have categories: x, y, z or xyz.
- `moveCheck`:
    - `Parameters`:
        - `self`
        - `category`: `str`, moving category.
        - `distList`: `list` of `int`, position list.
        - `msg`: `str='move'`
    - Check whether the mover has get to the desired destination, else continue moving.
- `move`:
    - `Parameters`:
        - `self`
        - `category`: `str`, moving category ('X', 'Y', 'Z', 'XY', 'XYZ').
        - `*dist`: `list` of `int`, moving destination position list.
    - Move to the ABSOLUTE coordinates, move category x, y, z, xy, xyz.
- `moveRela2Abs`:
    - `Parameters`:
        - `self`
        - `category`: `str`, moving category ('X', 'Y', 'Z', ' ').
        - `*relaDistList`: `list` of `int`, relative moving destination position list.
    - Move to RELATIVE coordinates, move category x, y, z, xy, xyz, but the algorithm is changing the relative input to absolute values
- `moveRela`:
    - `Parameters`:
        - `self`
        - `category`: `str`, moving category ('X', 'Y', 'Z').
        - `relaDist`: `int`, relative moving destination
    - Move to RELATIVE coordinates, move category x, y, z, but the algorithm is just moving relatively.

AmScope

Documentation (SDK): download

Current CCD version: AmScope MU2003-Bi. For colormode, Liguo has left enough space for changing from `BGR24` to `BGR48` for better color identification.

`class AmScope`:

- `__init__`:
    - `Parameters`:
        - `self`
        - `shortSleepTime`: `float=0.01`, set sleep interval (short) for programs.

        - `colormode`: `str='BGR24'`, set colormode.
        - `Temp`, `Tint`: `int=4448`, `int=843`, set temperature of color and style. You can see the effects through the AmScope software or `focus` in `scanGUI`.
        - `exposureTime`: `int=25000`, the unit is not the same as s, you could see the range through the AmScope software or `focus` in `scanGUI` (current is the largest).
        - `debug`: `bool=False`, set to `True` for debug process.
    - Set colormode, temperature of ccd color, exposure time of ccd camera.
- `snap`:
    - `Parameters`:
        - `self`
        - `res`: `str='full'`, resolution mode of ccd ('full': 5440x3648 pixels, 'half': 2736x1824 pixels, 'third': 1824x1216 pixels).
        - `colormode`: `str='BGR24'`, set color mode.
    - Take an image with corresponding resolution ('full', 'half', 'third')
- `open_camera`:
    - `Parameter`:
        - `self`
    - Open camera, and set the parameters of camera (buffer size, set trigger mode, set color temperature, set exposure time, white balance, auto exposure etc.)
- `set_resolution`:
    - `Parameters`:
        - `self`
        - `res`: `str='full'`, resolution mode of ccd ('full': 5440x3648 pixels, 'half': 2736x1824 pixels, 'third': 1824x1216 pixels).
        - `colormode`: `str='BGR24'`, set color mode.
    - Set resolution of cameras.
- `startPullCallback`:
    - `Parameter`:
        - `self`
    - Set Pull mode to snap image.
- `cameraCallback`:
    - `Parameters`:
        - `nEvent`
        - `ctx`
- `CameraCallback`:
    - `Parameters`:
        - `self`
        - `nEvent`
    - The vast majority of callbacks come from amcam.dll/so/dylib internal threads.

- `close_camera`:
  - `Parameter`:
    - `self`
- `__del__`:
  - `Parameter`:
    - `self`
  - Clear objects.

microscope

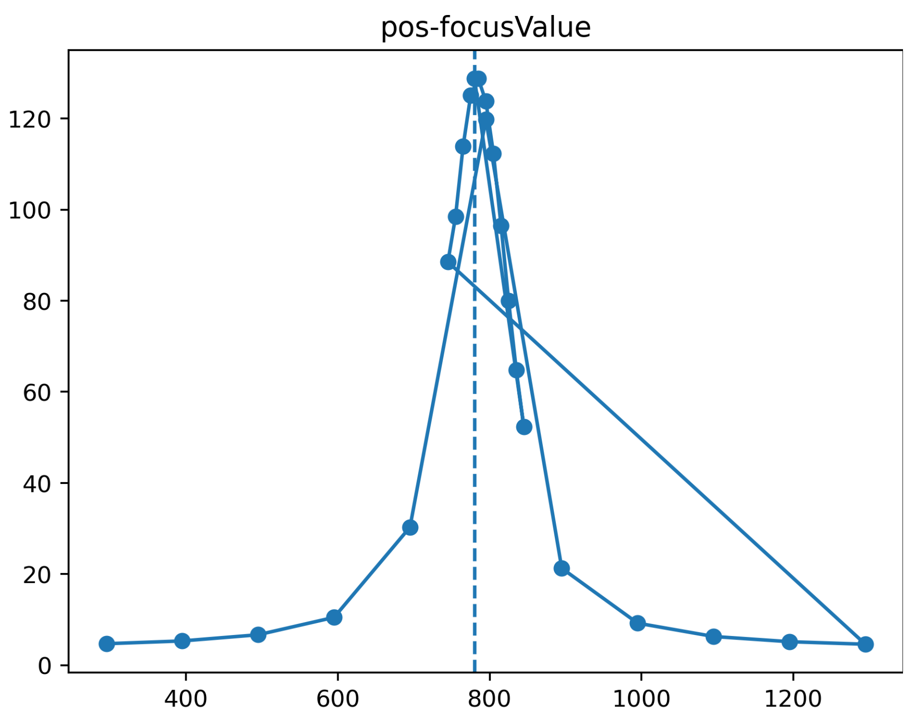
Import instance of `AmScope` and `priorMover`. Control two hardwares. Could control the stage and objective (through `priorMover`) and snap and save image (through `AmScope`).

`class microscope`:

- `__init__`:
  - `Parameter`:
    - `debug`: `bool = False`, set to `True` for debug process.
  - Create microscope instance, with opening AmScope and priorMover.
- `__del__`
  - `Parameter`:
    - `self`
  - Delete instance.
- `imgSnap`:
  - `Parameters`:
    - `self`
    - `res`: `str='full'`, resolution mode of ccd ('full', 'half', 'third').
  - `Return`:
    - `img`: snap image.
  - Return snap image.
- `threadImgSave`:
  - `Parameters`:
    - `self`
    - `img`: image data need to be saved.
    - `header`: `str`, image name.
    - `flatField`: `np.array=None`. Imported flatfield image data.
    - `isSaving`: `bool=True`, determine whether or not saving the image.
  - `Return`:
    - `bool`, `isSaving`.
  - A new thread for image saving.
- `imgSave`:

    - `Parameters`:
        - `self`
        - `img`: image data need to be saved.
        - `header`: `str`, image name.
        - `flatField`: `np.array=None`. Imported flatfield image data.
        - `isSaving`: `bool=True`, determine whether or not saving the image.
    - `Return`:
        - `bool`, `isSaving`.
    - Open a new thread for image saving (could be commented to other types).
- `focusValue`:
    - `Parameters`:
        - `self`
        - `img`: image data to be computed.
    - `Return`:
        - `float`, focus degree of an image.
    - Use laplacian to judge the focus degree of an image.
- `getCoor`:
    - `Parameter`:
        - `self`
    - `Return`:
        - `list` of `int`, coordinate values.
    - Get coordinate (x, y, z) of current position.
- `setCoor`:
    - `Parameters`:
        - `self`
        - `x`: `int=None`, x moving step.
        - `y`: `int=None`, y moving step.
        - `z`: `int=None`, z moving step.
    - Set ABSOLUTE destination for moving (type: x, y, xy, xyz).
- `setRelaCoor`:
    - `Parameters`:
        - `self`
        - `x`: `int=None`, x relative moving step.
        - `y`: `int=None`, y relative moving step.
        - `z`: `int=None`, z relative moving step.
    - Set RELATIVE destination for moving (type: x, y, xy, xyz).
- `plotImg`:

    - `Parameters`:
        - `self`
        - `winName`: `str`, name of opencv image window.
        - `img`: image data for show.
        - `title`: `str`, words need to put on images.
    - Plot image with opencv image windows.
- `autoFocus`:



Current algorithm: Hill climbing algorithm.

Just go in one direction and find the maximum position of one searching process, then go back around this position with finer steps to get best results of approximation.

-
    - `Parameters`:
        - `self`
        - `initPos`: `int=0`, initial absolute z position.
        - `relaPos`: `int=-2000`, initial relative z position, set current z position as origin.
        - `initStep`: `int=100`, initial step length, unit is preset.
        - `repeatTime`: `init=40`, repeat time for moving in one direction (a single searching process).
        - `fineRatio`: `int=20`, ratio to make the step finer. In each iteration, `initStep = initStep // fineRatio`
        - `isRela`: `bool=True`, determine whether the z position mode is relative (True) or absolute (False).
        - `header`: `str=''`, filename for output image.
        - `thresStep`: `int=6`, minimum threshold for step length.
        - `res`: `str='half'`, resolution mode of ccd ('full', 'half', 'third').
    - Automatically focus the objective.

sample

Store the information of a single substrate information, like corner information, focal plane, background color etc. .

`class sample`:

- `__init__`:
  - `Parameters`:
    - `self`
    - `initPos`: `list` of `int=[0,0]`, central position of the substrate (will finally be the correct value).
    - `focusPlane`: `list` of `float=[0,0,0]`, parameters for focal plane $z(x,y) = k_1x + k_2y + k_3$
    - `corner`: `np.array=np.zeros((4, 2))`, corner position values of substrate corners.
    - `xStep`, `yStep`: `int=1744//2`, `int=1079//2`, single step values for moving xy stage. Should have some pixels as overlap.
    - `folderName`: `str='new folder'`, folder name for saving substrate scan images.
    - `flatFieldPath`: `str='scanPart\\third.png'`, flatfield image path.
  - Create instance of a single substrate.
- `isSubstrate`:
  - `Parameters`:
    - `self`
    - `averPixel`: `np.ndarray`, input RGB value for judge.
    - `thres`: `float=100`, threshold distance for `averPixel` and `self.color`
  - `Return`:
    - `bool`: `True` if the distance between `averPixel` and `self.color` is less than `thres`, `False` otherwise.
  - Judge whether input RGB value belongs to substrate.
- `isSubstrateMat`:
  - `Parameters`:
    - `self`
    - `mat`: `np.array`
    - `thres`: `float=35`, threshold distance for `mat` and `self.color`
  - `Return`:
    - `bool`: `True` if the distance between `mat` and `self.color` is less than `thres`, `False` otherwise.
  - Compare all the pixels in `mat` to compare with `self.color` and find the minimal distance of pixels in mat with self.color, then compare it with `thres`. Thus if there are some defects but with still certain bare substrate parts, then it could still be identified correctly (**average RGB changes but the minimum distance will not be influenced a**

lot).

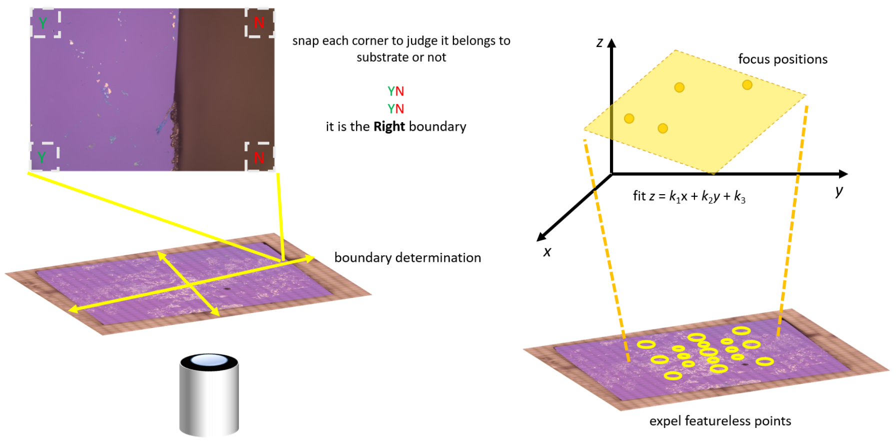
- `getFocusPos`:
    - `Parameters`:
        - `x`, `y`: `int`, focus z position of (x,y) using focal plane equation.
    - `Return`:
        - `int`, focus z position of (x,y) using focal plane equation.
    - Get focal position of (x,y)
- `modifyFocusPlane`:
    - `Parameters`:
        - `self`
        - `x0`, `y0`, `z0`: `int`
    - Modify focal plane equation for a new data point (x0,y0,z0) with just modifying k_3.
- flatfield image

Current: `third.png`.

Just use light with constant intensity to illuminate a featureless flat substrate. Make sure the resolution is right as your are scanning.

Note: **The light intensity is very import**. Modifying light intensity may induce false flatfield, and even make the trained model fail. So **be careful to fix the light intensity**. I recommend keeping the microscope at ccd mode and do not adjust the light intensity wheel.

sampleInit


Y
N
Y
N
snap each corner to judge it belongs to substrate or not
YN
YN
it is the **Right** boundary
boundary determination
z
focus positions
fit $z = k_1x + k_2y + k_3$
y
x
expel featureless points

Init instance of sample. Find the boundary, corner information, background and focal plane of a substrate.

`channels3Aver`:

- `Parameter`:
    - `mat`: `np.array`, input map of color with RGB values.
- `Return`:
    - `np.array`, mean value of three color channels in `mat`.
- Compute mean value of three color channels in a matrix.

`cornerCheck`:

- `Parameters`:
  - `sample`: instance of `sample`, input sample information.
  - `img`: `np.array`, scanned image from ccd.
  - `snapLen`: `int=300`, snap length of determination of a scanned image (whether belongs to substrate or vacuum).
- `Return`:
  - `str`, the analyzed positional result of the input `img`.
- Check corner category of a scanned image.

`bgColor`:

- `Parameters`:
  - `Parameters`:
    - `img`: `np.ndarray`, input image to be computed.
    - `radius`: `int`, the radius range of judging background color.
  - `Return`:
    - `np.ndarray`, the mean value of the background color.
  - Find the background color of an image.

`boundaryApproaching`:

- `Parameters`:
  - `microscope`: instance of `microscope`, control stage and objective.
  - `sample`: instance of `sample`, input sample information.
  - `category`: `str`, the category of approaching boundaries ('LEFT', 'RIGHT', 'UP', 'DOWN')
- Approach boundaries (left, right, up, down).

`cornerFinding`:

- `Parameters`:
  - `microscope`: instance of `microscope`, control stage and objective.
  - `sample`: instance of `sample`, input sample information.
  - `initPos`: `int=0`, initial absolute z position.
  - `relaPos`: `int=-800`, initial relative z position, set current z position as origin.
  - `initStep`: `int=100`, initial step length, unit is preset.
  - `repeatTime`: `init=16`, repeat time for moving in one direction (a single searching process).
  - `fineRatio`: `int=10`, ratio to make the step finer. In each iteration, `initStep = initStep // fineRatio`
  - `isRela`: `bool=True`, determine whether the z position mode is relative (True) or absolute (False).
  - `extraheader`: `str=''`, filename for output image.

    - `focusScale`: `float=0.5`, control the position of the selected focus points to the boundaries (0-1). The smaller the value is, the closer the focus points are to the boundaries.
- `Returns`:
    - `corner`: `np.ndarray`, `list` of corner position information.
    - `center`: `np.ndarray`, center position values.
    - `focusCorner[:,:3]`: `np.ndarray`, focus point x, y, z values.
- Find corners of a substrate, get background color, then find points to focus, and compute the focal plane for these points

`planeFunc`:

- `Parameters`:
    - `coor`: `np.array`, `list` of position (x,y) values
    - `k1`, `k2`, `k3`: `float`, parameters of focal plane
- `Return`:
    - `float`, value of fit z position using k_{1-3}
- Fit function for `scipy.optimize.curve_fit`.

`planeFit`:

- `Parameters`:
    - `outMat`: `np.ndarray`, focus point positions (x,y,z)
- `Return`:
    - `pOpt`: `np.ndarray`, parameters of focal plane
- Use `curve_fit` to find the best fit parameters for focal plane.

`centerRelaPos`:

- `Parameters`:
    - `xStep`, `yStep`: `int`, single step values for moving xy stage. Should have some pixels as overlap.
    - `corner`: `np.ndarray`, corner position information.
    - `center`: `np.ndarray`, center position values.
- `Return`:
    - `newCorner`: `np.ndarray`, relative integer positions reffered to the center position, divided with xStep and yStep.
- Compute relative scan position to the `center` (divide `xStep` and `yStep`, integers).

`corner2FocusPlane`:

- `Parameters`:
    - `corner`: `np.ndarray`, corner position information.
    - `center`: `np.ndarray`, center position values.
- `Return`:

    - `np.ndarray`, fit focal plane parameters
- Combine `corner` and `center` information as a whole matrix, then fit focal plane.

`corner2Boundary`:

- `Parameters`:
    - `xStep`, `yStep`: `int`, single step values for moving xy stage. Should have some pixels as overlap.
    - `corner`: `np.ndarray`, corner position information.
    - `center`: `np.ndarray`, center position values.
    - `expandScale`: `int=1`. extended length for dealing with not regular horizontal rectangular shapes.
- `Return`:
    - `np.ndarray`, scan matrix.
- Transform `corner`, `center`, `xStep`, `yStep` information to scan matrix.

`sampleInit`:

- `Parameters`:
    - `microscope`: instance of `microscope`, control stage and objective.
    - `sample`: instance of `sample`, input sample information.
- Overall process function of `sampleInit` part. Initialize the `sample` parameters.

chipScan

Use the scan matrix to scan the substrate.

`chipScan`:

- `Parameters`:
    - `microscope`: instance of `microscope`, control stage and objective.
    - `sample`: instance of `sample`, input sample information.

additional part

`amcam.dll`, `amcam.py`, `focusUi.py` are from official AmScope resources.

Detect Part

In parallel mode, we just run `scanPart` and `detectPart` simultaneously.

detector

`class MaterialDetector`:

The 2D material detector of the 2nd Insitute of Physics A, RWTH Aachen University.

The implementation is based on the following paper:

["An open-source robust machine learning platform for real-time detection and classification of 2D material flakes"]

- `__init__`:

    - `Parameters`:
        - `self`
        - `contrast_dict`: `dict`. The contrast dictionary of the material, Keys are the layer names, values are the contrast and the covariance matrix
        - `size_threshold`: `int`, optional. The minimal size of a flake in pixels. Defaults to 1000, this is about 281 $\mu\mathrm{m}^2$ in a 20x image.
        - `used_channels`: `str`, optional. The used channels for the detection. Defaults to "BGR" meaning all channels are used, BG would mean only the Blue and Green channel.
        - `false_positive_detector_path`: `str=None`, optional. The path to the false positive detector model. Use shape information to judge the possibility of a flake belongs to 2D materials or not.
    - Initialize a `MaterialDetector` instance.
- `_get_used_channel_indexes`:
    - `Parameters`:
        - `self`
    - `Return`:
        - `list[int]`, the indexes of the used channels.
    - Interprets the used_channels string and returns the indexes of the used channels. An example: `"BGR" -> [0,1,2]`; `"GR" -> [1,2]`.
- `_try_loading_fp_detector`:
    - `Parameters`:
        - `self`
        - `path`: `str`, the path of false positive detector model. Use shape information to judge the possibility of a flake belongs to 2D materials or not.
    - Try to load false positive detector model.
- `get_mean_background_values_numba`:
    - `Parameters`:
        - `image`: `NxMx3`, `np.array`. The image to calculate the mean background values from.
        - `radius`: `int`, optional. The size of the area around the mode of the histogram used for the calculations. Defaults to 5.
        - `min_value`: `int`, optional. The minimum value of the histogram used for the calculations, everything under this value will not be used. Defaults to 20.
        - `max_value`: `int`, optional. The maximum value of the histogram used for the calculations, everything above this value will not be used. Defaults to 230.
    - `Return`:
        - `np.ndarray`, the mean background values for each channel in form BGR, `dtype=np.uint8`.
    - Calculate the mean background values for each channel. Take the mean around the mode of the histogram of the image.

- `calculate_contrast_image`:
  - `Parameters`:
    - `image`: `HxWx3`, `np.array`, the image to calculate the contrast image.
    - `mean_background_values`: `NxMx3`, `np.array`, the mean background values for each channel in form BGR, `dtype=np.uint8`
  - `Return`:
    - `contrast_image`: `HxWx3`, `np.array`, the contrast image of the image.
  - Calculate the contrast image from the image and the mean background values. Sped up by using `numba`.
- `_get_fp_probability`:
  - `Parameters`:
    - `self`
    - `flake_contour`: `np.ndarray`, a `opencv` contour of the flake.
  - `Return`:
    - `float`, the probability of the flake being a false positive (0-1).
  - Calculate the probability of the flake being a false positive (not a real flake, could be like residues or cracks). Use the False Positive Detector.
- `_get_mean_entropy`:
  - `Parameters`:
    - `self`
    - `image`: `np.ndarray`, the original image.
    - `masked_flake`: `np.ndarray`, the mask of the flake.
    - `flake_contour`: `np.ndarray`, the `opencv` contour of the flake.
  - `Return`:
    - `float`, the mean shannon entropy of the flake.
- `_generate_mh_distance_map_from_contrast_image`:
  - `Parameters`:
    - `contrast_image`: `np.ndarray`, the image of shape `HxWxK`, `dtype=np.uint8`.
    - `means`: `np.ndarray`, the means of the Gaussian Mixture with `K` components.
    - `inv_choleskys`: `np.ndarray`, the inverse of the cholesky decomposition of the covariance matrix of the Gaussian Mixture with `K` components.
- `generate_mh_distance_map_from_contrast_image`:
  - `Parameters`:
    - `self`
    - `contrast_image`: `np.ndarray`, the contrast image of shape `HxWxK`, `dtype=np.uint8`.
  - `Return`:

        - `np.ndarray`, an array of shape (`KxHxW`) with `K` being the number of components and `H` and `W` being the height and width of the image. The values of the array are the Mahalanobis Distances of the pixels to the components.
    - Generate the Mahalanobis Distance Map of the Contrast image given the Gaussian Mixture Componentes. If you want to directly get the MH Distance Map you should call `generate_mh_distance_map` with the original image.
- `postprocess_mh_map`:
    - `Parameters`:
        - `self`
        - `distance_map`: `np.ndarray`, the Mahalanobis distance map of the image of shape (`KxHxW`) with `K` being the number of components and `H` and `W` being the height and width of the image
        - `distance_threshold`: `float`, optional, the Maximum Distance a value can have in Standard deviation. Defaults to 5.
    - `Return`:
        - `np.ndarray`, the semantic map of the flakes of shape (`KxHxW`) with `K` being the number of components and H and W being the height and width of the image.
    - Postprocess the Mahalanobis distance map to get the semantic map of the flakes. This generates a semantic map of flakes with no overlap.
- `detect_flakes`:
    - `Parameters`:
        - `self`
        - `image`: `NxMx3`, `np.array`, the original image without vignette, Expected to be in format BGR.
    - `Return`:
        - `Kx1` `np.array`, an array of flakes. See `flakeClass` for details.
    - Detect flakes in the given image. Expect images without vignette (being flatfielded)

flakeClass

`class Flake`:

This class is used to store the information of a flake.

- `__init__`:
    - `Parameters`:
        - `mask`: `np.ndarray`, the mask of the flake, a 2D array with 1s and 0s indicating the flake and background respectively.
        - `thickness`: `str`, the name of the layer the flake is from.
        - `size`: `int`, the size of the flake in pixels.
        - `mean_contrast`: `np.ndarray`, the mean contrast of the flake in BGR.
        - `center`: `tuple`, the center of the flake in pixels relative to the top left corner of the image.

        - `max_sidelength`: `int`, the maximum sidelength of the flake in pixels, measured using a rotated bounding box.
        - `min_sidelength`: `int`, the minimum sidelength of the flake in pixels, measured using a rotated bounding box.
        - `false_positive_probability`: `float`, optional. The probability of the flake being a false positive. Defaults to 0.
        - `entropy`: `float`, optional. The Shannon entropy of the flake. Defaults to -1.
    - Initialize a flake object.
- `export`:
    - `Parameters`:
        - `self`
        - `imgName`: `str`, the name of the image the flake is from.
        - `imgPath`: `str`, the path to the image the flake is from.
    - `Return`:
        - `dict`: {`"name"`, `"imgPath"`, `"thickness"`, `"size"`, `"mean_contrast"`, `"center"`, `"max_sidelength"`, `"min_sidelength"`, `"false_positive_probability"`, `"entropy"`}.
    - Export flake information to dict version. **Omit mask information**

flakeVisualize

Visualize the flake information to contours in original images.

`flakeList2Img`:

- `Parameters`:
    - `flakeList`: `list` of `Flake` to be visulized.
    - `imgPath`: `str`, the path to the image the flake is from.
    - `exportDir`: `str`, the folder to export images with visualized flakes.
    - `img`: `np.array=None`. If `img` is `None`, then we read `img` from `imgPath`. Otherwise just use `img` data.
    - `contourColor`: `list=[255,0,0]`. Select blue as the contour color.
- `Return`:
    - `imgList`: `list` of `str` for the exported images.
- Visualize flake contour in images.

`flakeList2Json`:

- `Parameters`:
    - `flakeList`: `list` of `Flake` to be visulized.
    - `imgPath`: `str`, the path to the image the flake is from.
    - `exportDir`: `str`, the folder to export json files of visualized flakes.
- Export flake information (other than mask) to json file.

`exportFlakeList`:

- `Parameters`:
  - `flakelist`: `list` of `Flake` to be visulized.
  - `imgPath`: `str`, the path to the image the flake is from.
  - `exportDir`: `str`, the folder to export json files of visualized flakes and images with visualized flakes.
  - `img`: `np.array=None`. If `img` is `None`, then we read `img` from `imgPath`. Otherwise just use `img` data.
  - `probLowerThres`, `probHigherThres`: `float=None`, the limit of false_positive_probability. Optional pre-set could enable selection of flakes within the range of `probLowerThres` and `probHigherThres`.
- Export flake to both json and image file, with optional limit of false_positive_probability.

flakeOutput

Conbine the detection and exportion from scanned images to flakes.

`class FlakeOutput`:

- `__init__`:
  - `Parameters`:
    - `self`
    - `bgColor`: `np.array`, rough image background color for identification of substrate or vacuum.
    - `modelJsonPath`: `str`, path of model json file.
    - `shapeDetectorPath`: `str='shapeModel.joblib'`, model path for false_positive model.
    - `sizeThres`: `int=500`
    - `usedChannels`: `str='BGR'`
  - Initialize a `FlakeOutput` instance.
- `outPut`
  - `Parameters`:
    - `self`
    - `imgPath`: `str`, path of image to be detected.
    - `img`: `np.ndarray=None`, if `img` is `None`, then we read `img` from `imgPath`. Otherwise just use `img` data.
    - `thres`: `float=2.5`, the threshold of Mahalanobis distance. When it get larger, meaning there maybe more false images get detected but the tolerance it bigger. When it get smaller, there maybe less false images but the tolerance is tight. So you need to test the balance of such trade-off.
  - Detect image and export the detection information.

imgCheck

Check whether the image is fully within substrate or not.

`isSubstrate` and `cornerCheck` please refer to the function in `sample` and `sampleInit`.

additional part

`shapeModel.joblib`: trained data using geometrical feature to roughly estimate the probability of a "flake" being actually residue or something else. The trained model is from the original paper. I just use the same copy of it for rough estimation.

`json` files are the export model files. The compenents could be referred from function `export` in `flakeClass`.

Read Part

worldMap

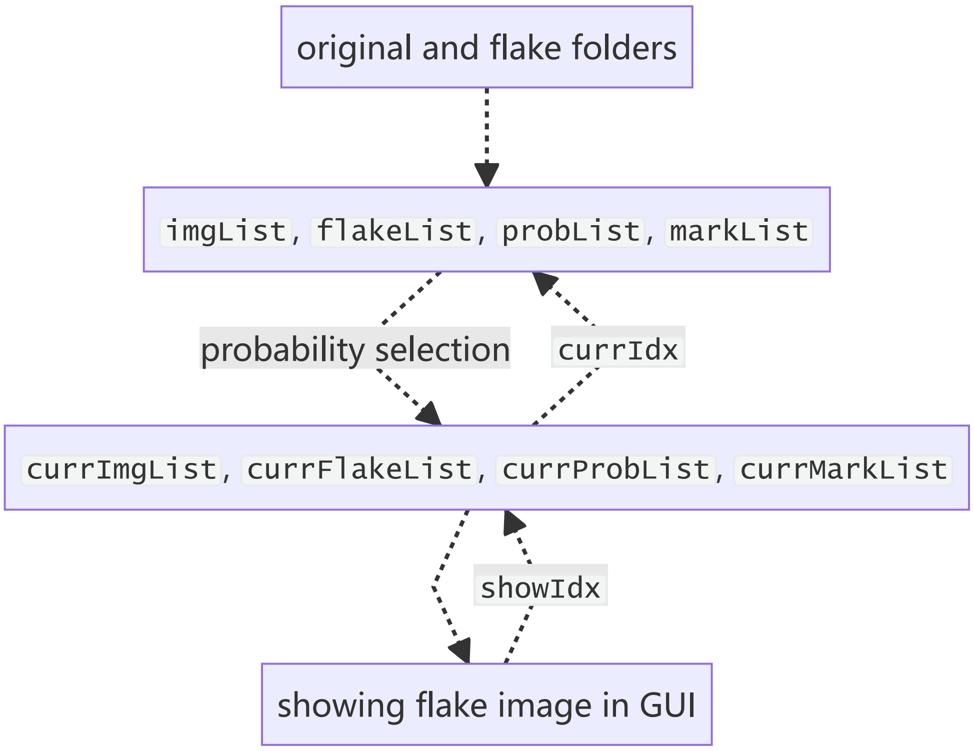
Generate worldmap after scanning the overall substrate images. Add contours to visualize flakes in worldmap as export data.

`class WorldMap`:

- `__init__`:
    - `Parameters`:
        - `self`
        - `imgDir`: `str`, folder path of the scanned images.
        - `contour`: `list` of positions of detected flakes.
        - `color`: `np.ndarray=np.array([255,0,0])`, the color of edge color of flakes. Now it is blue.
    - Initialize a `WorldMap` instance.
- `addMask`:
    - `Parameters`:
        - `self`
        - `imgPath`: `str`: path of the image having the flake of interest.
        - `contour`: `list`, `opencv` information of flake contours.
        - `color`: `np.ndarray=np.array([255,0,0])`, the color of edge color of flakes. Now it is blue.
    - `Return`:
        - `newWorldMap`: `np.ndarray`. The new worldmap with squares showing position of flakes.
    - Add flake contour to generate new worldmap output.
- `genWorldMap`:
    - `Parameters`:
        - `imgDir`: `str`, the image folder path.

        - `scale`: `float = 0.05`. The zoom-in ratio of image for combination. When enhance it, the file size of world map would be much smaller, but the resolution would be much smaller. Notice this trade off.
        - `cutXDim`, `cutYDim`: `int = 275`, `int = 300`. The cut pixel width of image overlap areas. You need to try to determine the exact number if the image is changed.
    - `Returns`:
        - `newMap`: `np.ndarray`, the worldmap of the scanned images in the folder.
        - `imgList`: `list` of `str`, the image name list of all the valid scanned images in the folder.
    - Generate a worldmap with corresponding scale information.

flakeLoad


original and flake folders
imgList, flakeList, probList, markList
probability selection
currIdx
currImgList, currFlakeList, currProbList, currMarkList
showIdx
showing flake image in GUI

`class FlakeLoad`:

- `__init__`:
    - `Parameters`:
        - `self`
        - `imgDir`: `str`, original image folder.
        - `loadDir`: str, check image folder (the folder with flake contours marked).
    - Initialize a `FlakeLoad` instance.
- `changeContent`:
    - `Parameters`:
        - `self`
        - `increment`: `int`, (-1,1).

        `Return`:

        - `bool`, `True` if the `showIdx` is valid. Otherwise `False`.
    - Switch to previous and next flake image.
- `probSelect`:
    - `Parameters`:
        - `self`
        - `probLowerThres`, `probHigherThres`: `float`, the limit of false_positive_probability. Optional pre-set could enable selection of flakes within the range of `probLowerThres` and `probHigherThres`.
    - Select flake images within the false_positive_probability range.
- `markSelect`:
    - `Parameters`:
        - `self`
        - `isSelect`: `bool`. `True` if the flake image is marked. Outherwise `False`.
    - Mark/unmark current flake image.
- `delCurr`:
    - `Parameters`:
        - `self`
        - `currDir`: `str`, folder path for flake images to be deleted.
    - `Return`:
        - `bool`. `True` if the flake image is deleted successfully. Otherwise `False`.
    - Delete current flake image
- `delUnMarked`:
    - `Parameters`:
        - `self`
        - `currDir`: `str`, folder path for flake images to be deleted.
    - `Return`:
        - `bool`. `True` if the flake images are deleted successfully. Otherwise `False`.
    - Delete all flake images that are not marked.
- `genFlakeWorldMap`:
    - `Parameters`:
        - `self`
        - `flake`: `dict` from flake json file.
        - `flakeImg`: original flake image.
        - `saveDir`: `str`, folder path for output flake image.
        - `maskColor`: `np.ndarray=np.array([255,0,0])`, the color of edge color of flakes (already masked). Now it is blue.
        - `labelColor`: `np.ndarray=np.array([255,0,0])`, the color of edge color of flakes (to be exported). Now it is blue.
    - `Return`:

        - `True` if flake image is exported successfully. Otherwise `False`.
    - Export single flake image with worldmap.
- `genCurrWorldMap`:
    - `Parameters`:
        - `self`
        - `saveDir`: `str`, folder path for output flake image.
    - `Return`:
        - `True` if flake image is exported successfully. Otherwise `False`.
    - Export current showing flake image with worldmap.
- `genMarkedWorldMap`:
    - `Parameters`:
        - `self`
        - `saveDir`: `str`, folder path for output flake image.
    - `Return`:
        - `True` if flake image is exported successfully. Otherwise `False`.
    - Export current marked flake image with worldmap.
- `genFlakeTrainDat`:
    - `Parameters`:
        - `self`
        - `flake`: `dict` from flake json file.
        - `trainDir`: `str`, folder path for output flake image for dataset.
        - `maskDir`: `str`, folder path for output masks of flake images.
        - `maskColor`: `np.ndarray=np.array([255,0,0])`, the color of edge color of flakes (already masked). Now it is blue.
        - `labelColor`: `np.ndarray=np.array([255,255,255])`, the color of flakes in masks. Now it is white.
    - `Return`:
        - `True` if flake image and mask image are exported successfully. Otherwise `False`.
    - Export single flake image with mask as train dataset.
- `genCurrTrainDat`:
    - `Parameters`:
        - `self`
        - `trainDir`: `str`, folder path for output flake image for dataset.
        - `maskDir`: `str`, folder path for output masks of flake images.
    - `Return`:
        - `True` if flake image and mask image are exported successfully. Otherwise `False`.
    - Export current showing flake image with mask as train dataset.
- `genMarkedTrainDat`:

        - `Parameters`:
            - `self`
            - `trainDir`: `str`, folder path for output flake image for dataset.
            - `maskDir`: `str`, folder path for output masks of flake images.
        - `Return`:
            - `True` if flake image and mask image are exported successfully. Otherwise `False`.
        - Export current marked flake image with mask as train dataset.

form and ui

GUI for image detection. The main functions are the same as `flakeLoad`. Please check the shortcuts in **Short Read Guide**.

The micron per pixel length in x20 objective is roughly 0.5343137254901961.

Main Folder

Main folder includes the files for training, detection and scanning. It use the scripts in previous folders.

train

`flatfield applying to folder`

- `Parameters`:
    - `flatfieldImgPath`: `str`, the file path of the fully illuminated image.
    - `imgDir`: `str`, the folder path of images need to be flatfielded.
    - `saveDir`: `str = None`, the folder path of output flatfielded images, if it is `None`, will automatically create a new folder which is `imgDir`'s name + `FlatField`.
- If you want to make the images in dirs flatfielded, apply it.
- In current version, we have done this process in scanning process, so you can just skip it. Please check carefully about the data you collected

`json information to mask generation`

- `Parameters`:
    - `jsonDir`: `str`, the folder path of `.json` file ready to be transformed.
    - `saveDir`: `str = None`, the folder path of output mask images, if it is `None`, will automatically create a new folder which is `jsonDir`'s name + `Mask`.
- `Return`:
    - `saveDir`: `str` when `saveDir` is not `None`, or return `jsonDir`'s name + `Mask`.
- The annotation tool we use is `labelme`, where you use polygonal to circle the flake area you what to set as examples. However, after annotation, it could only generate `.json` file, which records the edge and image information. To utilize that data in our codes, we need to transform it to mask images, where the flake area have different pixel values than other areas. This is the process we are doing: transfer the `.json` in annotation folder to another independent folder of mask images.

- **Note**: in some cases, like clarifying TMD layers, the categories could be determined by number of TMD layers (like monolayer or bilayer). However, in graphite or hBN, whose categories are mainly determined by exact numerical thickness values (like 3um or something else). In that case, there maybe issues about how to divide the clustering (or in other words, how many categories should we divide them since their thicknesses are so continuous). I suggest using thicks in `annotation` process:
    - When using `labelme`, you should have determined the thickness selection method (like you decided to divide them into categories like around 5um, 10um, 15um ...), then you text the thickness label in `labelme` (like 5, 10, 15 ...).
- Normally, the labels in `.json` file are useless since often we just cluster the pixels from the ground up. Currently, there is a file 'label.txt' in `saveDir`, counting all the labels you have used in annotation. I think in future we could use them for:
    1. the number of labels to determine the number of clusters
    2. the names of clusters, making them look more friendly

`get contrast from data points`

- `Parameters`:
    - `image_directory`: `str`, the folder path of images.
    - `mask_directory`: `str`, the folder path of mask images.
    - `lowerBound`, `upperBound`: `int`, the lower and higher bounds of mask image (`lowerBound < mask < upperBound`). They could be used to select certain value of thickness from mask (mask's pixel values correspond to the labels in annotation). Thus you could even just train one kind of thickness.
    - `isBoundSubtract`: `bool = False`. If you what to use `lowerBound` and `upperBound`, then you should set it as `True`.
- `Return`:
    - `dataContrast`: `np.ndarray`. Pix contrast in `np.ndarray` form, with BGR channels.
- The mask images record the areas you have selected. Then using them, you could get the pixel values of the areas you have selected. Here we compute the contrast using following formula:

$$\text{pix contrast} = \frac{\text{pix RGB value}}{\text{background RGB value}}$$

- **Note**: due to the intrinsic algorithm, if you find flake images which contain the edges of substrates, you would better not include that in your dataset, since the background colors may have errors.

`correlation heatmap plot & automatic data points crop`

- `Parameters`:
    - `data`: `np.ndarray`, the contrast data, you could just load from previous `dataContrast`.
    - `axis_names`: `list` of `str = ['Blue Contrast', 'Green Contrast', 'Red Contrast']`.
    - `sigma`: `float = 3`, σ for gaussian filter of data.
    - `bins`: `int = 200`, set the number of 'bins' in histogram.

  - `imgScale`: `int = 1`, set the size of image and fonts. If you find the output does not suit the screen, you can modify it.
  - `process_function`: `= lambda x: np.log(x+1)`, just for final image visualization effect.
  - `upper_bounds`, `lower_bounds`: `list = []`, if you want to use self-defined `upper_bounds` and `lower_bounds` to filter data points, you could make them both not `None`.
  - `title`: `str = 'Full 3D Contrast Heatmap`. Feel free to change it!
  - `used_channels`: `str = 'BGR'`, set the used color channels, you could use 'BG' or something else if you find only two channels are strongly correlated.
  - `autoCropRatio`: `float = 1.5`. The automatic crop ratio of data points, which sets the cutoff value as $\text{mean value of contrast} \ / \ \text{autoCropRatio}$. Feel free to change it if you find 1.5 is not suitable.
- `Returns`:
  - `dataCropped`: `np.ndarray`. The new contrast data points using autoCrop. You can omit it if you don't want to use it.
  - `boundMin`, `boundMax`: `np.ndarray`. Record the autoCrop bounds in `array` forms.
- Show the heatmap of contrast data points. The orange dashed lines will show the autoCropped data points area.

`gaussian k-means clustering for contrast data`

- `Parameters`:
  - `data`: `np.ndarray`, previous contrast data points. You could use original or cropped data.
  - `num_components`: `int`, number of gaussian clustering components.
  - `cov_type`: `str = 'full`. You could select from `{'full', 'tied', 'diag', 'spherical'}`. Details could be referred from `sklearn` documentation.
  - `num_additional_noise_comp`: `int = 0`, number of noise components. **Note**: if you use `num_additional_noise_comp`, it will dismiss some gaussian clusters as noise. Thus `num_components` should be `num_components` + `num_additional_noise_comp`
  - `sample_size`: `int = 30000`, upper limit of sampled data points. When the data set number is larger than that, it will randomly sample `sample_size` data points.
  - `used_channels`: `list` of `str`, same as that in `correlation heatmap plot & automatic data points crop`
  - `initial_means`: `list`, the initial values for gaussian cluster. Details could be referred from `sklearn` documentation.
  - `**kwargs`: `**kwargs` for `sklearn.GaussianMixture`. Details could be referred from `sklearn` documentation.
- `Returns`:
  - `all_means_gauss`, `all_covariances_gauss`, `all_weights_gauss`, `predicted_labels`: means, covariance matrices, weights and predicted labels of gaussian clusters. Details could be referred from `sklearn` documentation
- Use `sklearn` to do gaussian clustering.

`draw gaussian fit data`

- `Parameters`:
    - `data`: `np.ndarray`, previous contrast data points. You could use original or cropped data.
    - `predict_labels`, `gauss_means`, `gauss_weights`, `gauss_covariances`: from `gaussian k-means clustering for contrast data`.
    - `lower_bounds`, `upper_bounds`: `list = []`. Manually crop the data points. Same as that in `gaussian k-means clustering for contrast data`
    - `axis_names`: `list = ['Blue Contrast', 'Green Contrast', 'Red Contrast']`. Same as that in `gaussian k-means clustering for contrast data`
    - `heatmap_sigma`: `float = 2`. σ of gaussian filter.
    - `heatmap_bins`: `int = 100`, bins for heatmap histograms.
    - `plot_type`: `str, {'scatter', 'heatmap'}`, default = `'heatmap'`. Plot image type.
    - `bins`: `int = 50`, bins for edge histograms.
    - `fig_size`: `tuple = (3, 3)`, size of figures.
    - `font_size`: `int = 5`, size of fonts.
    - `used_channels`: `str = 'BGR`, same as that in `correlation heatmap plot & automatic data points crop`
- `Returns`:
    - `figures`, `axes`: `plt.figure`
- Draw heatmap, with ellipse of confidence and histograms with gaussian fit curves.

`export json file`

- `Parameters`:
    - `gaussMean`, `gaussCov`: from `gaussian k-means clustering for contrast data`
    - `saveName`: `str`, the name of output `.json` file
    - `saveDir`: `str = None`, if it is `None`, will generate `json` file at the same folder of `train.ipynb`.
- Generate `.json` file for gaussian clusters.

detect

`folder detection`

- `Parameters`:
    - `path`: `str`, the image folder path.
    - `bgColor`: `np.array=None`. It is the information stored in scanning folder. So normally you do not need to input it.
    - `modelJsonPath`: `str`, the model json path.
    - `shapeDetectorPath`: `str`, the shape detector joblib path.
    - `sizeThres`: `int = 500`, the pixel limit for detection. Current length relation is about $1 \text{ pixel} \approx 0.53\mu\text{m}$.

    - `usedChannels`: str = 'BGR', set the used color channels, you could use 'BG' or something else if you find only two channels are strongly correlated.
    - `scale`: `float = 0.1`. The zoom-out ratio of the scanned image for combination of images.
    - `cutXDim`, `cutYDim`: `int = 100`, `int = 100`. The cut pixel width of image overlap areas. You need to try to determine the exact number if the image is changed.
    - `thres`: `float = 2.5`, the threshold of Mahalanobis distance. When it get larger, meaning there maybe more false images get detected but the tolerance it bigger. When it get smaller, there maybe less false images but the tolerance is tight. So you need to test the balance of such trade-off.
- Manually detect the folder of scanning data.

`world map generation`

- `Parameters`:
    - `imgDir`: `str`, the image folder path.
    - `scale`: `float = 0.05`. The zoom-in ratio of image for combination. When enhance it, the file size of world map would be much smaller, but the resolution would be much smaller. Notice this trade off.
    - `cutXDim`, `cutYDim`: `int = 275`, `int = 300`. The cut pixel width of image overlap areas. You need to try to determine the exact number if the image is changed.
- `Returns`:
    - `newMap`: `np.array`, output world map.
    - `imgList`: `list[str]`, list of all the image file names for combination.
- Generate world map of a scanning image folder. You could try multiple times especially when you want to get best value of overlap cut length in x and y dimensions.

find

`class codeEventHandler`:

- `__init__`:
    - `Parameters`:
        - `self`
        - `shortSleepTime`: `float = 0.05`, set sleep interval (short) for programs.
        - `longSleepTime`: `float = 0.1`, set sleep interval (long) for programs.
        - `bgColor`: `np.array=None`. It is the information stored in scanning folder. So normally you do not need to input it.
        - `modelJsonPath`: `str`, the model json path.
        - `shapeDetectorPath`: `str`, optional. The path to the false positive detector model. Use shape information to judge the possibility of a flake belongs to 2D materials or not.
        - `sizeThres`: `int`, optional. The minimal size of a flake in pixels. Defaults to 500, this is about 140 $\mu\mathrm{m}^2$ in a 20x image.

        - `scale`: `float = 0.1`. The zoom-in ratio of image for combination. When enhance it, the file size of world map would be much smaller, but the resolution would be much smaller. Notice this trade off.
        - `cutXDim`, `cutYDim`: `int = 100`, `int = 100`. The cut pixel width of image overlap areas. You need to try to determine the exact number if the image is changed.
        - `thres`: `float=2.5`, the threshold of Mahalanobis distance. When it get larger, meaning there maybe more false images get detected but the tolerance it bigger. When it get smaller, there maybe less false images but the tolerance is tight. So you need to test the balance of such trade-off.
    - Initialize a `codeEventHandler` instance.
- `on_created`:
    - `Parameters`:
        - `self`
        - `event`: `dict`, the information of newly created file.
    - Handle image file creation event, create a thread for its dection process.

`find`:

- `Parameters`:
    - `Path`: `str`, path of scan image folder.
    - `bgColor`: `np.array=None`. It is the information stored in scanning folder. So normally you do not need to input it.
    - `thres`: `float=2.5`, the threshold of Mahalanobis distance. When it get larger, meaning there maybe more false images get detected but the tolerance it bigger. When it get smaller, there maybe less false images but the tolerance is tight. So you need to test the balance of such trade-off.
    - `sizeThres`: `int`, optional. The minimal size of a flake in pixels. Defaults to 500, this is about 140 $\mu\mathrm{m}^2$ in a 20x image.
    - `modelJsonPath`: `str`, the model json path.
    - `shapeDetectorPath`: `str`, optional. The path to the false positive detector model. Use shape information to judge the possibility of a flake belongs to 2D materials or not.
    - `used_channels`: `str = 'BGR'`, set the used color channels, you could use 'BG' or something else if you find only two channels are strongly correlated.
    - `scale`: `float = 0.1`. The zoom-in ratio of image for combination. When enhance it, the file size of world map would be much smaller, but the resolution would be much smaller. Notice this trade off.
    - `cutXDim`, `cutYDim`: `int = 100`, `int = 100`. The cut pixel width of image overlap areas. You need to try to determine the exact number if the image is changed.
- Create `codeEventHandler`. Detect each newly scan (created) image.

`findDir`

- `Parameters`:
    - `path`: `str`, the image folder path.

    - `bgColor`: `np.array=None`. It is the information stored in scanning folder. So normally you do not need to input it.
    - `modelJsonPath`: `str`, the model json path.
    - `shapeDetectorPath`: `str`, the shape detector joblib path.
    - `sizeThres`: `int = 500`, the pixel limit for detection. Current length relation is about $1 \text{ pixel} \approx 0.53 \mu\text{m}$.
    - `usedChannels`: str = 'BGR', set the used color channels, you could use 'BG' or something else if you find only two channels are strongly correlated.
    - `scale`: `float = 0.1`. The zoom-out ratio of the scanned image for combination of images.
    - `cutXDim`, `cutYDim`: `int = 100`, `int = 100`. The cut pixel width of image overlap areas. You need to try to determine the exact number if the image is changed.
    - `thres`: `float = 2.5`, the threshold of Mahalanobis distance. When it get larger, meaning there maybe more false images get detected but the tolerance it bigger. When it get smaller, there maybe less false images but the tolerance is tight. So you need to test the balance of such trade-off.
- Manually detect the folder of scanning data.

RunFile

`class ScanRun`:

- `__init__`:
    - `Parameters`:
        - `self`
        - `folderName`: `str`, the folder name of the scan images.
        - `modelPath`: `str`, the model json path.
    - Initialize a `ScanRun` instance.
    - **Note**: if the position of vacuum box changes, `self.x0` and `self.y0` should be changed for the new ancher of the overall scan position.
- `__del__`:
    - `Parameters`:
        - `self`
- `scan`:
    - `Parameters`:
        - `self`
        - `x`, `y`: `int`, (x,y) for the position in vacuum box for substrate scanning.
        - `posName`: `str`, position name added for scan image folder in this substrate.
    - Scan the substrate in the position (x,y) and save the scan image to the folder. Open a parallel process to detect the substrate in the scan image.

scanGui and scanUi

GUI for setting scanning process.

Note: the geometry for scan positions on vacuum box is set in `class Widget` `setScanMat`. If the geometry changes, should refer to that part.

Basically, the scanning thread is a `QThread` and runs `scanRun.scan` parallelly. The scan status and time are shown based on the communication of `QThread` and main program with `pyqtSignal`. For details refer to the documentation of `pyQt5`.

`class Widget`:

- `setScanMat`:
  - `Parameter`:
    - `self`
  - Set the geometry for scan positions on vacuum box. There is relationship between the label name to real vacuum box positions. For example, `s1->1`, `s15->1.5` etc. . But checking the buttons clicked, we could set the scan matrix and the following folder names. Please check with **Short Scan Guide**.

Acknowledgment

Thanks for the collabraction with Dr. Liguo Ma and also the helpful discussions with Dr. Hongyuan Li and Dr. Kaifei Kang. This project is generously supported by Prof. Kin Fai Mak and Prof. Jie Shan.

If there are any extra issues, feel free to contact me through email or slack: ys2289@cornell.edu or sym20@mail.ustc.edu.cn.

Yiming Sun, Dec 12th, 2023

Univerisity of Science and Technology of China