autopew Documentation

Release 0.1.4

Morgan Williams & Louise Schoneveld

May 13, 2022

Getting Started

CHAPTER 1

Introduction

autopew is designed for arbitrary translation between planar/2D Cartesian coordinate systems using affine transforms and human-in-the-loop workflows.

This is applied to integrating coordinate systems across analytical instrumentation, with each instrument typically having its own individual coordinate systems based on imagery and/or a sample stage. **autopew** also includes functions for importing and exporting files, for automated generation of point sets within a relevant format for each piece of analytical instrumentation. **autopew** outputs currently included a .scancsv file which can be directly imported into Chromium laser ablation navigation software.

CHAPTER 2

Why use **autopew**

autopew is designed for easy referencing between analytical equipment and/or images. This allows the time spend on analytical equipment to be more effectively used for data collection rather than spending valuable time locating the areas of interest.

This software also allows for tracking of the context of in-situ microanalysis by allowing reference to large images and areas which will allow for new insights into what effects chemistry of given particles with reference to their location and micro-environment. We can then track the analysis between different analytical equipment and make inferences on macroscale processes from well characterised in-situ microanalysis[1].

Although primarily designed for use of laser ablation analysis on geological material this software can be used for any microanalytical technique, including electron microprobe analysis, x-ray fluorescence mapping, scanning electron microscopy and ion beam analysis.

See also:

For outlined examples of how autopew is used, see Examples

[1] Pearce, M. A., Godel, B. M., Fisher, L. A., Schoneveld, L. E., Cleverly, J. S., Oliver, N. H. S., and Nugus, M. (2017). Microscale data to macroscale processes: a review of microcharacterization applied to mineral systems: Geological Society, London, Special Publications,, v. 453 doi:10.1144/SP453.3.

CHAPTER 3

What is **autopew** not?

- Not currently capable of 3D affine transforms (i.e. no ‘focus’ attribute).

The current development plan for **autopew** can be found here.

CHAPTER 4

Getting Started

4.1 Getting Started

Note: This page is under construction. Feel free to send through suggestions or questions.

4.1.1 Set up

To use this software you need in install Python. Anaconda is a great way to install python and included popular software for editing and interacting with python script such as Spyder and Jupyter.

4.1.2 Installation

See also:

Installation

For in depth installation processes see the page above. For most purposes you should be able to install **autopew** by typing the following in a terminal ("Anaconda Prompt" application on your computer if you're using Anaconda)

```
pip install autopew
```

If you're already installed autopew and would like the most up-to-date version, type this into a terminal:

```
pip install --upgrade autopew
```

4.1.3 Writing and editing code

autopew is designed to be very beginner friendly, but if you're completely new to python consider visiting some free online courses about the basic Python concepts. Codecademy is a great jumping off point.

4.2 Installation

autopew is available on PyPi, and can be downloaded with pip:

```
pip install autopew
```

Note: autopew is not yet packaged for Anaconda, and as such `conda install autopew` will not work.

4.2.1 Upgrading autopew

New versions of pyrolite are released frequently. You can upgrade to the latest edition on PyPi using the `--upgrade` flag:

```
pip install --upgrade autopew
```

4.3 Development Installation

autopew is a work in progress. The development version is in a public repository on GitHub. You can install it using pip directly from there:

```
pip install git+https://github.com/morganjwilliams/autopew.git#egg=autopew
# or, for the develop version
pip install git+https://github.com/morganjwilliams/autopew.git@develop#egg=autopew
```

Alternatively, you can also clone it locally and install with pip:

```
git clone https://github.com/morganjwilliams/autopew.git
cd autopew
# e.g if you navigate to the directory
pip install .
```

If you want to contribute to autopew, you might want to use an editable installation locally to debug:

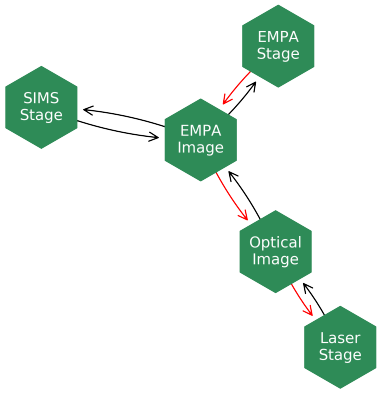
```
git clone https://github.com/morganjwilliams/autopew.git
cd autopew
# e.g if you navigate to the directory
pip install -e .[dev]
```

CHAPTER 5

Examples

5.1 Use Cases

autopew is designed for easy referencing between analytical equipment and/or images. This allows users to easily transfer between techniques such as electron probe, laser ablation, scanning electron microscope or other imaging techniques. autopew will allow consistent measurements of the same grains via different techniques and give spatial context to chemical data.



There are a number of use cases that autopew is suited for:

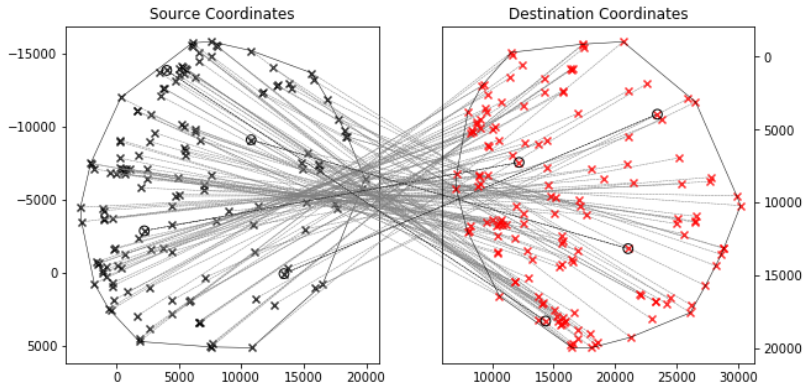
5.1.1 Two Sets of Points

E.g. coordinates from stages This case is for transfer between various stage coordinate system (e.g. electron probe and laser ablation) which can allow for measurement of the same grain with different techniques.

What you need:

- X,Y coordinates of the points you wish to analyse

- at least 3 points in the new coordinate system



autopew can translate points with rotation, and shear.

See also:

stage2stage workflow

5.1.2 Image and Set of Points

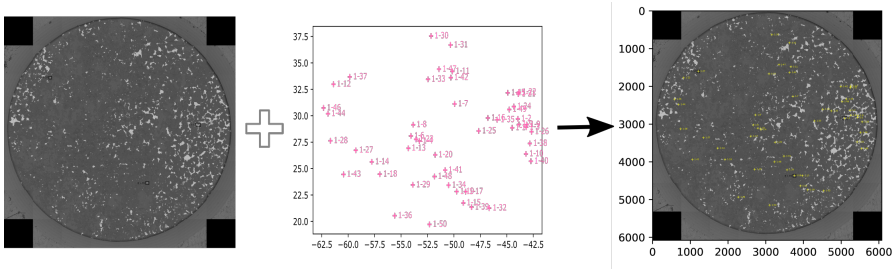
If you have high resolution microscope images or images from other sources such as X-ray fluorescence mapping you can use pixel coordinates and convert the chosen points to the new stage coordinates.

This allows you to pick phases for analysis before your analytical session without reducing the analytical time wasted on locating and programming the coordinates of the phases of interest.

You can also mark analytical locations on a large image to give context information to the microanalysis by converting stage coordinates to pixel coordinates.

What you need:

- an image of high enough resolution to identify the target phases
- 3 points in the new coordinate system that you can recognise on the image



See also:

image to stage workflow, stage to image workflow

5.1.3 Two Images

in development

See the contributions page on how to contribute.

This allows the pixel coordinates from one image to be translated into the pixel coordinates in a second image. This is useful if you need to overlay two images such as an x-ray fluorescence image over a reflected light images

What you need:

- two images you wish to overlay
- 3 features you can recognise on both images

5.2 Examples

Here are some examples of what **autopew** can do and how you can do it. Before using these examples, please visit the installation and getting started pages.

See also:

Installation , Getting Started

5.2.1 Workflows

This set of pages walks you through how to use **autopew** for each use case.

Image to Stage

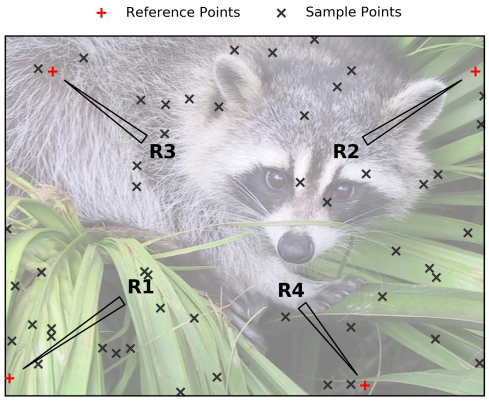
This is a simple workflow example involving converting points registered on an image to coordinates for a specific stage.

INPUT: image of sample

OUTPUT: coordinates for use in navigating sample

e.g we have been using this method to extract phases from reflected light imagery or X-ray fluorescence mapping and assign numbers to grains to give spatial context to the multiple different micro-geochemical analysis.

Input and output can be easily changed for your purposes see the contributions page for more information on how to contribute.



Step 1: Acquire an Image and Register Points

- Aquire an image of your sample
- add points to your image

Once an image is acquired, points can be added using **autopew** directly or using external software (e.g. ImageJ or Fiji*[0]). If you use ImageJ, export your points as a .csv file and follow the Stage to stage workflow which outlines transforming a list of X,Y coordinates into a new translation. The following workflow is designed using the **autopew** extensions.

For selection points directly in **autopew** here is an example:

Step 3: Calibrate the Transformation between the Image and Stage

- Pick a 3 or more calibration points

Note that the calibration of this transform involves a least-squares process to find the optimal transformation, such that adding more calibration points can help avoid minor inaccuracies in adding points.

Step 4: Transform Image Point Coordinates to Stage Coordinates

- give the same reference points in the stage Coordinates (in this case laser reference coordinates)
- Use **autopew** to transform all pixel coordinates to stage coordinates. See the example code below:

Step 5: Export Points to for Stage Coordinates

- Export the transformed point stage coordinates to a file you can import into the software controlling the stage.

```
import numpy as np
from pathlib import Path
from autopew import Pew
from autopew.workflow import pick_points

# %% PICK THE ANALYSIS POINTS FROM AN IMAGE ---------------------------------
↪---------------------------
# have an image you wish to use?
imagepath = Path("./../../source/_static/") / "img.jpg"

# pick sample coordinates from the image
Sample_points = pick_points(imagepath)

# %% REFERENCE POINTS -------------------------------------------------------
↪-----
#these are the known locations of the reference points on the laser stage
laser_REF_coords = np.array([
      [74978,85419], #R1
      [80259,75389], #R2
      [90828,82571], #R3
      [81465,74373]]) #R4
```

(continues on next page)

[0] Schneider, C. A.; Rasband, W. S. & Eliceiri, K. W. (2012), "NIH Image to ImageJ: 25 years of image analysis", Nature methods 9(7): 671-675, PMID 22930834

(continued from previous page)

```
# pick sample coordinates from the image
Sample_REF_points = pick_points(imagepath)

# %% TRANSFORM ----------------------------------------------------------
↪-
points = (Pew(Sample_REF_points,
              laser_REF_coords)
              .load_samples(Sample_points))
# %% VISUALISE ----------------------------------------------------------
from autopew.util.plot import plot_transform
fig = plot_transform(
    points.samples[['x', 'y']].values,
    points.transformed[['x', 'y']].values,
    invert0=[False, False],
    invert1=[False, False]
)

# %% EXPORT -------------------------------------------------------------
# lets save them so we can directly import them to the laser, with a known␣
↪focal length (20744)
points.export_samples("samples.scancsv", z=20744)
```

See also:

output types

Optional Next Steps

- Export an aligned image.

Imported images can be realigned to the stage coordinate system for easier recognition of sample features and more accurate visual determination of new point location. See stage to image

References

Stage to Image

This is a simple workflow example involving converting points from one stage coordinate system to display on a large image. This helps to visualise where the analysis were collected and gives context to the anaysis.

INPUT:

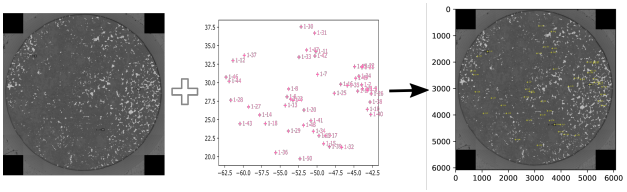
- .csv - a list of X,Y coordinates with spotnames + more than 3 reference points
- large image with locations of more than 3 reference points

OUTPUT:

- image with points labelled

e.g. We have been using the workflow to visualise the microanalysis points from SEM on large reflected light images

Input and output can be easily changed for your purposes see the contributions page for more information on how to contribute.



Step 1: Acquire image

- collect an image of the sample
- Remember to highlight 3 regions or more for registration points

Step 2: Calibrate and Transform the points between the the image and the stage

- Import your CSV file with analysed points
- specify your >3 reference coordinates using the **autopew** interactive interface
- specify the stage Coordinates of these reference points
- Use **autopew** to transform all stage coordinates. See the example code below:

Note that the calibration of this transform involves a least-squares process to find the optimal transformation, such that adding more calibration points can help avoid minor inaccuracies in adding points.

```
from pathlib import Path
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from autopew import Pew
from autopew.workflow import pick_points

imagepath = Path("./../../source/_static/") / "SEM_RefPoints.png"

# %% REFERENCE POINTS ------------------------------------------------------------
#these are the known locations of the reference points in the laser coordinate system
laser_reference_coords =np.array([[45633,9098],  #R1-ccp55
                                  [56683,17876], #R2-ccp33
                                  [43301,16082], #R3-ccp38
                                  [42096,5137]]) #R4-pn25

#pick the reference points on the image
img_reference_coords = pick_points(imagepath)

# %% TRANSFORM laser to pixels ---------------------------------------------------
↪--------
points = (Pew(laser_reference_coords,
              img_reference_coords)
              .load_samples('Samples.csv'))
```

Step 4: Overlay the image and the points

- Export an image containing labelled point overlay over image

```
# FIND THE PIXEL SIZE OF THE IMAGE
img = Image.open(imagepath)
# get the image's width and height in pixels
width, height = img.size

fig, ax = plt.subplots()
ax.scatter(points.transformed['x'], points.transformed['y'],facecolors='none',␣
↪edgecolors='y', marker="o",zorder=1,s=6,linewidth=.3)

for i, df in enumerate(points.transformed['name']):
  ax.annotate(df, (points.transformed.x[i], points.transformed.y[i]),
              xytext=(2, 0), textcoords='offset points',
              horizontalalignment='left', verticalalignment='center',
              size=4, color='yellow',
              zorder=1)
ax.set(xlim=(0, width), ylim=(0, height))

plt.imshow(img, zorder=0)
ax.invert_yaxis()#image invert so it is the same up direction as import.

plt.tight_layout()
plt.show()
#fig.savefig('Export_image.png', transparent=True, dpi=800)
```

See also:

output types

Stage to Stage

This is a simple workflow example involving converting points from one stage coordinate system to another. This workflow also works for importing .csv files of pixel x,y coordinates.

For this example we use:

INPUT: .csv - a list of x,y coordinates with names + more than 3 reference points

FORMAT: the transform requires column names of 'x' and 'y' and also recognises 'name' as spotnames

OUTPUT: pos, scancsv or csv file in laser coordinate system with corresponding spotnames

Input and output can be easily changed for your purposes see the contributions page for more information on how to contribute.

e.g. We have been using the workflow to ensure measurement of the same grain on both scanning electron microscope and the laser ablation system.

Step 1: Acquire coordinates

- save the coordinates of 3 or more registration points
- collect and save coordinates of phases of interest

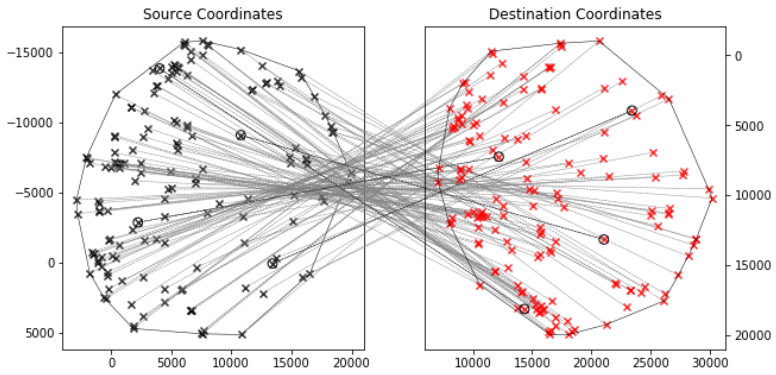
Step 2: Export Point Coordinates to CSV

- Export the stage-coordinates of your planned points in x-y format to a CSV.

Optionally, specify names for each of the points, which could be used as an index later.

Step 3: Calibrate and Transform the points between the two stages

- Use **autopew** to transform all stage coordinates. See the example code below:



Note that the calibration of this transform involves a least-squares process to find the optimal transformation, such that adding more calibration points can help avoid minor inaccuracies in adding points.

Step 4: Export Points to for new Stage Coordinates

- Export the transformed point stage coordinates to a file you can import into the software controlling the new stage.

```
from autopew import Pew
from autopew.util.plot import plot_transform

# %% LOAD reference points and sample points ----------------------------------------
↪----------------------
points = (Pew("Reference Points_source.csv",
              "Reference Points_dest.csv")
              .load_samples('Samples.csv'))

# %% VISUALISE THE TRANSFORM ---------------------------------------------------------
↪------
fig = plot_transform(
    points.samples[['x', 'y']].values,
    points.transformed[['x', 'y']].values)

# %% EXPORT --------------------------------------------------------------
points.export_samples("Points_new.pos")
points.export_samples("Points_new.scancsv")
points.export_samples("Points_new.csv")
```

See also:

output types

5.3 Output Types

The output types can be edited to suit the equipment you wish to use. The current output of coordinates are designed for the following analytical equipment.

5.3.1 Laser Ablation system

The .scancsv current output is compatible with a Photonmachines (Teledyne) laser running on Chromium. This allows for rapid input of X,Y coordinates into the laser system with spot labels either read from the source file, or numbered sequentially.

Currently all other settings; including fine focus (z), spot size and laser conditions need to be changed manually after import into the laser.

5.3.2 JEOL EPMA

The .pos files for the JEOL field-emission gun electron probe microanalyser (EPMA) using "probe for EPMA" software. A default z value can be assigned for each export however fine focus will need to be changed manually.

5.3.3 TESCAN SEM

Work in Progress

Use of the TESCAN SEM system allows input and output of .XML format coordinates. Currently only allows export and import of a single .xml file per sample. Multiple samples in a single .xml file is in development.

This export type does not include labels for points.

Focus is set per sample and needs to be manually adjusted for each analysis location.

5.3.4 Adding New IO Functionality

New file types can be configured around the `autopew.io.PewIOSpecification` interface, as have been done for the above instruments. These will be automatically registered as file handlers where the file extension is unique. For example, see the soure code specification for `autopew.io.PewSCANCSV` which includes a function for reading *.scancsv* files into a consistent format and outputting data from that format into *.scancsv* files.

CHAPTER 6

Development

6.1 API

6.1.1 autopew

autopew.Pew

class autopew.**Pew**(**args*, *transform=None*, *archive=None*, ***kwargs*)

calibrate(*src*, *dest*, *handler=None*, ***kwargs*)
Calibrate the transformation between two planar coordinate systems given two sets of corresponding points.

Parameters

- **src** (str | pathlib.Path | numpy.ndarray | pandas.DataFrame)
- **dest** (str | pathlib.Path | numpy.ndarray | pandas.DataFrame)
- **handler** (str | tuple)

export_samples(*filepath*, *enforce_transform=True*, ***kwargs*)
Export a set of coordinates.

Parameters

- **filepath** (str | pathlib.Path) – Desired export filepath.
- **enforce_transform** (bool) – Whether to enforce transformation before export.

load_samples(*filepath*, *handler=None*, ***kwargs*)
Import a set of sample coordinates.

Parameters **filepath** (str | pathlib.Path | numpy.ndarray | pandas.DataFrame)

Returns

Return type pandas.DataFrame

to_archive(*filepath*)
Archive the coordinate mapping and calibration for later loading.

Parameters **filepath** (str | pathlib.Path)

transform_samples(*samples=None*, *limits=None*, ***kwargs*)
Transform sample coordinates to the destination coordinate system.

Parameters **limits** (list | numpy.ndarray)

Returns

Return type numpy.ndarray

6.1.2 autopew.transform

Submodule for calculating and visualising affine transforms between planar coordinate systems.

autopew.transform

Submodule for calculating and visualising affine transforms between planar coordinate systems.

autopew.transform.**affine_from_AB**(*X*, *Y*)
Create an affine transforamtion matrix based on two sets of coordinates.

Note:

- This is an augmented matrix, and includes the translation component

autopew.transform.affine

Submodule for calculating affine transforms between planar coordinate systems.

autopew.transform.affine.**affine_from_AB**(*X*, *Y*)
Create an affine transforamtion matrix based on two sets of coordinates.

Note:

- This is an augmented matrix, and includes the translation component

autopew.transform.affine.**affine_transform**(*A*)
Create an affine transform function based on affine matrix A.

autopew.transform.affine.**compose_affine2d**(*T=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), Z=array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]), R=array([[1., 0., 0.], [-0., 1., 0.], [0., 0., 1.]])*)
Compose an affine transformation matrix based on translation, zoom and rotation components.

Parameters **T, Z, R** (numpy.ndarray) – Component affine transfrom matricies for translation, zoom/scaling and rotation.

Returns **A**

Return type numpy.ndarray

autopew.transform.affine.**corners**(*size*)

autopew.transform.affine.**decompose_affine2d**(*A*)
Decompose an affine transform into components using a polar transform.

Returns T, Z, R

Return type numpy.ndarray

Note: This decomposes the transform into the sequence rotation - zoom - translation.

To recompose this transform,

---

autopew.transform.affine.**rotate**(*theta=0*, *degrees=True*)
Generate a 2D affine rotation matrix.

Uses clockwise rotations.

autopew.transform.affine.**shear**(*x=0*, *y=0*)
Generate a 2D affine shear matrix.

autopew.transform.affine.**translate**(*x=0*, *y=0*)
Generate a 2D affine translation matrix.

autopew.transform.affine.**zoom**(*x=1*, *y=1*)
Generate a 2D affine zoom matrix.

autopew.transform.vis

autopew.transform.vis.**vis**(*A*, *ax=None*)
Visualise the effect of an affine transfrom on a unit square.

Parameters

- **A** (numpy.ndarray) – Affine matrix to visualise.
- **ax** (matplotlib.axes.Axes, *None*) – Axes to plot on.

Returns

Return type matplotlib.axes.Axes

6.1.3 autopew.io

File operations for autopew. Contains a class to provide an generalized interface to a series of file types/instruments which can be built upon to add and register new IO capability.

autopew.io

File operations for autopew. Contains a class to provide an generalized interface to a series of file types/instruments which can be built upon to add and register new IO capability.

class autopew.io.**PewIOSpecification**(**args*, ***kwargs*)
Template for input and output file handlers for autopew.

These handers specify functions to import files to pandas DataFrames and the export of these file-types from pandas DataFrames.

extension = None

type = None

classmethod validate_dataframe(*df*)
Validate the output of a file reader against the minimum requirements for autopew.
Parameters **df** (pandas.DataFrame) – Dataframe to validate.

class autopew.io.**PewCSV**(**args*, ***kwargs*)

extension = '.csv'

classmethod read(*filepath*, ***kwargs*)

classmethod write(*df*, *filepath*, ***kwargs*)

class autopew.io.**PewSCANCSV**(**args*, ***kwargs*)

extension = '.scancsv'

classmethod read(*filepath*)

classmethod write(*df*, *filepath*, ***kwargs*)

class autopew.io.**PewJEOLpos**(**args*, ***kwargs*)

extension = '.pos'

classmethod write(*df*, *filepath*, ***kwargs*)

autopew.io.**registered_extensions**()
Get a dictionary of registered extensions mapping the relevant IO specifications.

Returns

Return type dict

autopew.io.laser.chromium

Import and export functions for the Chromium Laser Ablation Navigation Software.

autopew.io.laser.chromium.**get_scandata**(*scandict*)
Process a dictionary of scan information into a DataFrame.

Parameters **scandict** (dict) – Dictionary of scan data.

Returns

Return type pandas.DataFrame

See also:

ScanData

autopew.io.laser.chromium.**read_lasefile**(*filepath*, *encoding='cp1252'*)
Read a .lase formatted file into a dictonary (one item per scan), and return this in the form of a DataFrame.

Parameters

- **filepath** (str, pathlib.Path) – Path to the .lase file to import.
- **encoding** (str) – File encoding of the .lase file.

Returns

Return type DataFrame

Notes

.lase files have all spot information as well as gas flows and all control options of laser operation. These are configuration files structured by blocks demarkated with square brackets, along the lines of:

```
[Scans]
Header=Scan Type,Description, ...
Scan0=Spot,"Spot 1", ...
[<other control blocks>]
...
```

Some of the values in this table are string-encoded tuples (e.g. *"57950.00,37530.00,20734.50"*) and dictionaries (e.g. *"Dosage=1;DwellTime=1.00;LineSpacing=100.00;Laser.Output=10.00;..."*).

See also:

ScanData, *get_scandata()*

autopew.io.laser.chromium.**read_scancsv**(*filepath*, *encoding='cp1252'*)

Read a .scancsv into a dictonary (one item per scan), and return this in the form of a DataFrame.

Parameters

- **filepath** (str, pathlib.Path) – Path to the .scancsv file to import.
- **encoding** (str) – File encoding of the .scancsv file.

Returns

Return type DataFrame

Notes

.scancsv files only contain spot information and do not edit the laser and gas conditions. These are essentially comma-separated values files, with a structure along the lines of:

```
Scan Type,Description, ...
Spot,"Spot 1", ...
```

Some of the values in this table are string-encoded tuples (e.g. *"57950.00,37530.00,20734.50"*) and dictionaries (e.g. *"Dosage=1;DwellTime=1.00;LineSpacing=100.00;Laser.Output=10.00;..."*).

See also:

ScanData, *get_scandata()*

autopew.io.laser.chromium.**split_config**(*s*)

Splits a config-formatted string.

Parameters **s** (str)

Returns

Return type dict

See also:

get_scandata()

autopew.io.laser.chromium.**write_scancsv**(*df*, *filepath=PosixPath('exportedpoints.scancsv')*, *spotnames=None*, *encoding='cp1252'*, *z=20800*, ***kwargs*)

Export an array of coordinates to a .scancsv file.

Parameters

- **df** (pandas.DataFrame) – Dataframe containing points to serialise.
- **filepath** (str | pathlib.Path) – Filepath for export.
- **spotnames** (str | list) – Name to prefix spot indicies or a list of spot names.
- **encoding** (str) – Encoding for the output file.
- **z** (int) – Optional specification of default focus value to use.

Returns

Return type pandas.DataFrame

autopew.io.EPMA.JEOL

Export function for the JEOL field-emission gun electron probe microanalyser (EPMA) using "probe for EPMA".

autopew.io.EPMA.JEOL.**write_pos**(*df*, *filepath=PosixPath('exportedpoints.pos')*, *encoding='cp1252'*, *z=10.7*, ***kwargs*)

Export an dataframe of coordinates to a .pos file.

Parameters

- **df** (pandas.DataFrame) – Dataframe containing points to serialise.
- **filepath** (str | pathlib.Path) – Filepath for export.
- **encoding** (str) – Encoding for the output file.
- **z** (int) – Optional specification of default focus value to use.

Returns

Return type pandas.DataFrame

6.1.4 autopew.gui

autopew.gui

Basic Graphical User Interface support for autopew, allowing interactive the picking of analysis points.

6.1.5 autopew.image

Submoudle for working with image data, used in interactive elements of autopew.

autopew.image

Submoudle for working with image data, used in interactive elements of autopew.

class autopew.image.**PewImage**(*img*, *extent=None*, *transform=None*)

affine_transform(*A*, *resample=0*, *reversey=True*, *reverserotation=False*)
Transform the image via an affine transformation matrix using PIL.

load_image(*img*)
Load an image and deal with formatting etc.

maprgb()
Convert the image to RGB arrays.

thumb(*frac=0.05*, *resample=<Resampling.BILINEAR: 2>*)
Return a thumbnail downsampled version of the image.

6.1.6 autopew.graph

Submodule for graph representation and use of chained affine transformations.

autopew.graph.network

class autopew.graph.network.**Net**
Network of transformations between objects.

This object stores the individual node objects including their properties and registration points, in addition to the edges which involve specific coordinate transforms.

add_edge(*A*, *B*, *transform=None*, ***kwargs*)
Add an edge between components A and B. Optionally specify the specific transform.

Parameters
- **A, B** (str) – Names of components to link.
- **transform** (function) – Function to transform coordinates from A space to B space.
- **inverse_transform** (function) – Function to transform coordinates from B space to A space.

draw(*ec='k'*, *nc='seagreen'*, *ax=None*, *figsize=(10, 10)*, *method=<function draw_shell>*)

edges

get_transform(*A*, *B*)
Get the function to transform coordinates between nodes A and B.

link(*A*, *B*, *transform=<function autolink>*, *inverse_transform=<function autolink>*, ***kwargs*)
Link nodes A and B with transforms along edges.

nodes

update(*name*, *obj*, ***kwargs*)

autopew.graph.network.**autolink**(*x*)
Default link function.

6.1.7 autopew.util

Submodule containing a small set of utilities for autopew.

autopew.util.plot

autopew.util.plot.**bin_centres_to_edges**(*centres*)
Translates point estimates at the centres of bins to equivalent edges, for the case of evenly spaced bins.

autopew.util.plot.**bin_edges_to_centres**(*edges*)
Translates edges of histogram bins to bin centres.

autopew.util.plot.**data_to_pixels_transform**(*ax*)

autopew.util.plot.**plot_2dhull**(*data*, *ax=None*, *s=0*, ***plotkwargs*)
Plots a 2D convex hull around an array of xy data points.

autopew.util.plot.**plot_transform**(*src*, *dest=None*, *tfm=None*, *ref=None*, *ax=None*, *sharex=False*, *sharey=False*, *invert0=[False, True]*, *invert1=[False, True]*, *figsize=(10, 5)*, *titles=['Source Coordinates', 'Destination Coordinates']*, *hull=True*)
Visualise an affine transfrom.

autopew.util.meta

autopew.util.meta.**autopew_datafolder**(*subfolder=None*)
Returns the path of the autopew data folder.

Parameters **subfolder** (str) – Subfolder within the autopew data folder.

Returns

Return type pathlib.Path

autopew.util.meta.**chain**(*lst*)
Chain a series of fuctions together.

6.2 Development

autopew is currently hosted on GitHub at github.com/morganjwilliams/autopew; collaborator access can be granted to interested parties. If you're new to Git or GitHub, there are some useful guides on the GitHub Website.

6.2.1 Development Installation

To access and use the development version, you can either clone the repository or install via pip directly from GitHub:

```
pip install git+git://github.com/morganjwilliams/autopew.git@develop#egg=autopew
```

6.2.2 Branches and GitFlow

There are two main git-branches for `autopew`:

- `master` is the latest stable release.
- `develop` is the development branch.

The Git workflow is based on GitFlow, where releases are branched from `develop` prior to being integrated into `master`. Pull requests should be made against the `develop` branch.

6.2.3 Documentation

Documentation is currently live on *ReadtheDocs.org <https://autopew.readthedocs.io>`__* , but can also be built and viewed locally using instructions below. The documentation is built using sphinx, and most pages are written in reStructuredText. A quick reference can be found here.

Documentation for `autopew` is in the `docs` directory. From this directory, documentation can be built as follows:

To build documentation on windows:

```
# to build and view the html version:
make html && cd ./build/html/ && index.html && cd ../..
# or, to build and view the latex-pdf version:
make latex && cd ./build/latex/ && make.bat && autopew.pdf && cd ../..
```

Alternatively, there is a default build batch file `makeviewhtml.bat` also located in the docs directory, which executes the commands above and will automatically build the docs and open the landing page:

```
# to build and view the html version:
makeviewhtml.bat
```

6.2.4 Tests

If you clone the source repository, unit tests can be run using pytest from the root directory after installation:

```
python setup.py test
```

6.2.5 Continuous Integration

There are also some active continuous integration tools for `autopew`, including automated unit-testing on Travis-CI and test coverage analysis on Codecov. The details for each of these are listed below.

Travis-CI

The Travis-CI page for `autopew` can be found at travis-ci.org/morganjwilliams/autopew.

Code Coverage

The coveralls page for `autopew` can be found at coveralls.io/github/morganjwilliams/autopew.

6.3 Changelog

All notable changes to this project will be documented here.

6.3.1 Development

Note: Changes noted in this subsection are to be released in the next version. If you're keen to check something out before its released, you can use a development install.

6.3.2 0.1.3

6.3.3 0.1.2

6.3.4 0.1.1

- Expanded development documentation.
- Updated installation instructions.
- Added basic documentation examples and a workflow runthrough.
- Added a network-based transformation concept example.

`autopew.transform`

- Added `autopew.transform.CoordinateTransform`, inverse affine transform

6.3.5 *0.1.0* (Unreleased)

6.3.6 0.0.2

- Added PyQT requirement for GUI-based point-picking.

`autopew.gui`

- Update for GUI point selection to add refreshing timeout.
- Renamed `image_registration()` to `image_point_registration()`; later moved to `autopew.util.gui.image_point_registration()`
- Added differentiated handling of mouse events for panning, zooming and clicking in `autopew.util.gui`

`autopew.registration`

- Updated `autopew.registration.RegisteredImage` image handling to allow load from path/array/existing image.
- Added `set_calibration_pixelpoints()` for setting calibration points for a registered image.

autopew.session

- Added `load_image()`, `points_from_csv()`, `autoflow()` and stubs for `reorder_analyses()`, `standard_bracket()` (neither implemented in this version).
- Added an automated workflow for export of coordinates from a CSV, image and stage coordinates in `autoflow()`.

autopew.transform

- Added a *rcond* switch for `numpy.linalg.lstsq()` for *Python <= 3.6* in `autopew.transform.calibration` due to recurring errors.

autopew.util

- Added `autopew.util.readlase` for reading specific laser analysis files.

6.3.7 0.0.1

- First version of the package, with capability for basic point-point and image-point calibration/registration.
- Added submodules `autopew.session`, `autopew.targets`, *autopew.gui*, `autopew.transform.calibration`, `autopew.registration`, `autopew.util`
- Added some basic tests.

6.4 Future

This page details some of the under-development and planned features for **autopew**. Note that while no schedules are attached, features under development are likely to be completed with weeks to months, while those 'On The Horizon' may be significantly further away (or in some cases may not make it to release).

Initial development focuses on work relevant to LA-ICP-MS workflows, and the project may later generalise some of these workflows.

6.4.1 Current Release

- Able to register points on one image
- Able to calibrate these points to the laser stage coordinates
- Able to output the image with points on it
- Able to output a .scancsv file to import into the laser software
- Able to import point names

6.4.2 Under Development

- Be able to set spotsizes
- Be able to read and set z position

- Be able to recognise when point is positioned outside stage limits
- Be able to transform maps with points on them so correspondence is easier to see
- Should be able to work from multiple maps –> need multiple registered images
- Overlay images (see post here)
- Serialising coordinate transform systems for later use

6.4.3 On The Horizon

- Activities to optimise instrument usage time:
  - Sample-standard bracketing using a specific 'reference mount location' (needs to be updated later)
  - Reordering points based on their positions.
  - Standard area registration, gridded 'free positions'- auto sample standard bracketing
- Report templates
- Dealing with larger images (>20 MB - jpg will warn of compression bomb)

6.5 Code of Conduct

6.5.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

6.5.2 Our Standards

Examples of behaviour that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behaviour by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

6.5.3 Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behaviour and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behaviour.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviours that they deem inappropriate, threatening, offensive, or harmful.

6.5.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

6.5.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behaviour may be reported by contacting the project admins. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

6.5.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant Version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html. The Contributor Covenant is released under the Creative Commons Attribution 4.0 License.

For answers to common questions about this code of conduct, see https://www.contributor-covenant.org/faq.

6.6 Contributing

`autopew` welcomes community contributions of all forms. Requests for features and bug reports are particularly valuable contributions, in addition to code and expanding the documentation.

All individuals contributing to the project are expected to follow the Code of Conduct, which outlines community expectations and responsibilities.

Also, be sure to add your name or GitHub username to the contributors list.

Note: This project is currently in *alpha*, and as such there's much work to be done.

6.6.1 Feature Requests

If you're new to Python, and want to implement something as part of `autopew`, you can submit a Feature Request. Perhaps also check the Issues Board first to see if someone else has suggested something similar (or if something is in development), and comment there.

6.6.2 Bug Reports

If you've tried to do something with `autopew`, but it didn't work, and googling error messages didn't help (or, if the error messages are full of `autopew.XX.xx`), you can submit a Bug Report . Perhaps also check the Issues Board first to see if someone else is having the same issue, and comment there.

6.6.3 Contributing to Documentation

The documentation and examples for `autopew` are gradually being developed, and any contributions or corrections would be greatly appreciated. Currently the examples are patchy, and a 'getting started' guide would be a helpful addition. If you'd like to edit an existing page, the easiest way to get started is via the 'Edit on GitHub' links:

These pages serve multiple purposes:

- A human-readable reference of the source code (compiled from docstrings).
- A set of simple examples to demonstrate use and utility.
- A place for developing extended examples

6.6.4 Contributing Code

Code contributions are always welcome, whether it be small modifications or entire features. As the project gains momentum, check the Issues Board for outstanding issues, features under development. If you'd like to contribute, but you're not so experienced with Python, look for `good first issue` tags or email the maintainer for suggestions.

To contribute code, the place to start will be forking the source for `autopew` from GitHub. Once forked, clone a local copy and from the repository directory you can install a development (editable) copy via `python setup.py develop`. To incorporate suggested changes back to into the project, push your changes to your remote fork, and then submit a pull request onto autopew/develop .

Note:

- See Installation for directions for installing extra dependencies for development, and Development for information on development environments and tests.
- `autopew` development roughly follows a gitflow workflow. `autopew/master` is only used for releases, and large separable features should be build on `feature` branches off `develop`.
- Contributions introducing new functions, classes or entire features should also include appropriate tests where possible (see *Writing Tests*, below).
- `autopew` uses Black for code formatting, and submissions which have passed through `Black` are appreciated, although not critical.

6.6.5 Writing Tests

There is currently a minimal unit test suite for `autopew`, which guards against breaking changes and assures baseline functionality. `autopew` uses continuous integration via Travis, where the full suite of tests are run for each commit and pull request, and test coverage output to Coveralls.

Adding or expanding tests is a helpful way to ensure `autopew` does what is meant to, and does it reproducibly. The unit test suite one critical component of the package, and necessary to enable sufficient trust to use `autopew` for scientific purposes.

6.7 Contributors

This list includes people who have contributed to the project in the form of code, comments, testing, bug reports, and feature requests.

- Louise Schoneveld
- Morgan Williams

Note: This documentation is a work in progress and is updated regularly. Contact the maintainers with any specific questions/requests.

CHAPTER 7

References

Python Module Index

a

Index

R

S

T

U

V

W

Z