DefDAP

Release 0.93.3dev

Michael D. Atkinson, Rhys Thomas, João Quinta da Fonseca

Nov 30, 2021

CONTENTS

DefDAP

Deformation Data Analysis in Python

DefDAP is a python library for correlating EBSD and HRDIC data. It was developed by Michael Atkinson and Rhys Thomas during their PhDs at the Univeristy of Manchester.

This documentation gives information about the latest version of DefDAP.

CHAPTER

ONE

CITATION

If this software is used in the preparation of published work please cite:

Atkinson, Michael D, Thomas, Rhys, Harte, Allan, Crowther, Peter, & Quinta da Fonseca, João. (2020, May 4). DefDAP: Deformation Data Analysis in Python - v0.92 (Version 0.92). Zenodo.

CHAPTER

TWO

LICENSES

This software is distributed under Apache License 2.0. For more details see the *License* page.

CHAPTER

THREE

CONTENTS

3.1 Installation

The defdap package is available from the Python Package Index (PyPI) and can be installed by executing the following command:

```
pip install defdap
```

The prerequisite packages should be installed automatically by pip, but if you want to manually install them, using the conda package manager for example, then run the following command:

```
conda install scipy numpy matplotlib scikit-image scikit-learn pandas networkx␣
↪jupyter ipython
```

If you are doing development work on the scripts, first clone the repository from GitHub. The package can then be installed in editable mode using pip with flag -e to create a "linked" .egg module, which means the module is loaded from the directory at runtime. This avoids having to reinstall every time changes are made. Run the following command from the root of the cloned repository:

```
pip install -e .
```

3.2 Contributing

We welcome any improvements made to this package, but please follow the guidelines below when making changes.

3.2.1 Coding style

In this project we (try) to follow the PEP8 standard for formatting python code with 2 notable exceptions: - Function and variable names are in mixed case e.g myFirstFunction - The 79 character line length limit is seen more as a guideline than rule. Code is split across lines where possible and to improve readability. Line length should never excede 119 characters. All documentation should be split to lines of less than 73 characters.

3.2.2 Repository Structure

The schematic below shows the intended structure of the branches in this repository, based on Gitflow. The *master* branch will be committed to at regular intervals by an administrator and each release will be given a tag with a version number. This is the first branch that is visible and we (will eventually) have controls in place to ensure this branch always works correctly. The *develop* branch contains new features which may not be significant enough to form part of the 'master' branch just yet. The final type of branch is the feature branch (green), such as *feture_1* and *feature_2*. This is where you come in! If you would like to make a new feature or fix a bug, simply make a branch from develop with a reasonable name and make a start on any changes you would like to make. Don't be afraid of breaking anything at this point, there are controls in place to help make sure buggy code does not make it into *develop* or *master*.

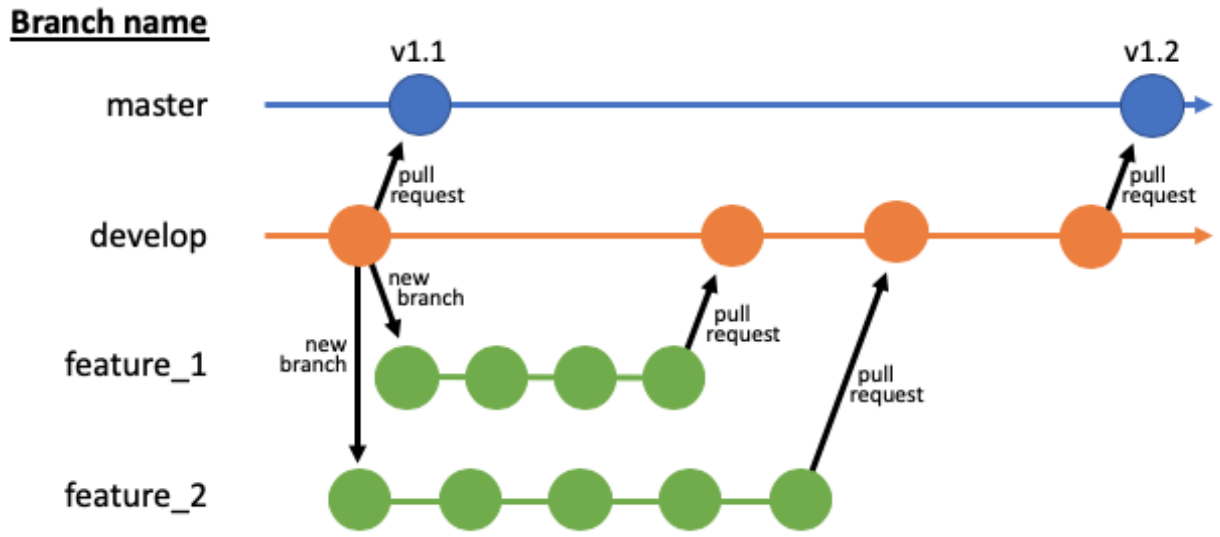

Fig. 1: Repository structure

3.2.3 Instructions

1. Making an 'issue' first is recommended when adding a new feature or fixing a bug, especially if you're not sure how to go about the change. Advice can then be given on the best way forward.
2. Using your local git client (GitHub Desktop is easy to use), checkout the *develop* branch by selecting it (making sure you 'Pull Origin' after).
3. Create a new branch with an appropriate name, using underscores where appropriate, for example *new_feature*
4. Make any changes necessary in your favourite editor (PyCharm is recommended).
5. Test to make sure the feature works correctly in your Jupyter Notebook.
6. Commit the change to your new branch using the 'Commit to new_feature' button and include a descriptive title and description for the commit, making sure you click 'Push origin' when this is done.
7. Make additional commits if necessary.
8. Raise a pull request.

3.2.4 Additional notes

- Always make a branch from *develop*, don't try to make a branch from *master* or any feature branch.
- You will not be able to complete a pull request into *develop* without a review by Mike or Rhys.
- Try to avoid adding any dependencies to the code, if you do need to, add a comment to your 'issue' with the details.

3.2.5 Documentation

Where possible, update or add documentation at the beginning of the function you are making or changing, adding references if required. This is important so that other people know how to use your code and so that we can validate any methods you use. We are following the NumPy Docs Style Guide, but you can use any of the documentation in the code as an example. Add comments where it is not clear what you have done.

3.3 Papers

Here is a list of papers which have use the DefDAP Python library.

3.3.1 2020

- R.Sperry, A.Harte, J.Quinta da Fonseca, E.R.Homer, R.H.Wagoner, D.T.Fullwood. Slip band characteristics in the presence of grain boundaries in nickel-based superalloy. Acta Materialia. Volume 193, July 2020, Pages 229-238.
- A.Harte, M.Atkinson, M.Preuss, J.Quinta da Fonseca. A statistical study of the relationship between plastic strain and lattice misorientation on the surface of a deformed Ni-based superalloy. Acta Materialia. May 2020.
- A.Harte, M.Atkinson, A.Smith, C.Drouven, S.Zaefferer, J.Quinta da Fonseca, M.Preuss. The effect of solid solution and gamma prime on the deformation modes in Ni-based superalloys. Acta Materialia. May 2020.
- D.Lunt, A.Ho, A.Davis, A.Harte, F.Martin, J.Quinta da Fonseca, P.Prangnell. The effect of loading direction on strain localisation in wire arc additively manufactured Ti–6Al–4V. Materials Science and Engineering: A. May 2020, 139608.
- D.Lunt, R.Thomas, M.Roy, J.Duff, M.Atkinson, P.Frankel, M.Preuss, J.Quinta da Fonseca. Comparison of sub-grain scale digital image correlation calculated using commercial and open-source software packages. Materials Characterization. Volume 163, May 2020, 110271.

3.3.2 2019

- X.Xu, D.Lunt, R.Thomas, R. Prasath Babu, A.Harte, M.Atkinson, J.Quinta da Fonseca, M.Preuss. Identification of active slip mode in a hexagonal material by correlative scanning electron microscopy. Acta Materialia. Volume 175, 15 August 2019, Pages 376-393.
- R.Thomas, D.Lunt, M.D.Atkinson, J.Quinta da Fonseca, M.Preuss, F.Barton, J.O'Hanlon, P.Frankel. Characterisation of irradiation enhanced strain localisation in a zirconium alloy. Materialia. Volume 5, March 2019, 100248.

3.3.3 2018

- D.Lunt, A.Orozco-Caballero, R.Thomas, P.Honniball, P.Frankel, M.Preuss, J.Quinta da Fonseca. Enabling high resolution strain mapping in zirconium alloys. Materials Characterization. Volume 139, May 2018, Pages 355-363.

3.4 License

(continues on next page)

(continued from previous page)

```
      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      copyright license to reproduce, prepare Derivative Works of,
      publicly display, publicly perform, sublicense, and distribute the
      Work and such Derivative Works in Source or Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have made,
      use, offer to sell, sell, import, and otherwise transfer the Work,
      where such license applies only to those patent claims licensable
      by such Contributor that are necessarily infringed by their
      Contribution(s) alone or by combination of their Contribution(s)
      with the Work to which such Contribution(s) was submitted. If You
      institute patent litigation against any entity (including a
      cross-claim or counterclaim in a lawsuit) alleging that the Work
      or a Contribution incorporated within the Work constitutes direct
      or contributory patent infringement, then any patent licenses
      granted to You under this License for that Work shall terminate
      as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and
```

(continues on next page)

(continued from previous page)

```
    (d) If the Work includes a "NOTICE" text file as part of its
        distribution, then any Derivative Works that You distribute must
        include a readable copy of the attribution notices contained
        within such NOTICE file, excluding those notices that do not
        pertain to any part of the Derivative Works, in at least one
        of the following places: within a NOTICE text file distributed
        as part of the Derivative Works; within the Source form or
        documentation, if provided along with the Derivative Works; or,
        within a display generated by the Derivative Works, if and
        wherever such third-party notices normally appear. The contents
        of the NOTICE file are for informational purposes only and
        do not modify the License. You may add Your own attribution
        notices within Derivative Works that You distribute, alongside
        or as an addendum to the NOTICE text from the Work, provided
        that such additional attribution notices cannot be construed
        as modifying the License.

    You may add Your own copyright statement to Your modifications and
    may provide additional or different license terms and conditions
    for use, reproduction, or distribution of Your modifications, or
    for any such Derivative Works as a whole, provided Your use,
    reproduction, and distribution of the Work otherwise complies with
    the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
    any Contribution intentionally submitted for inclusion in the Work
    by You to the Licensor shall be under the terms and conditions of
    this License, without any additional terms or conditions.
    Notwithstanding the above, nothing herein shall supersede or modify
    the terms of any separate license agreement you may have executed
    with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
    names, trademarks, service marks, or product names of the Licensor,
    except as required for reasonable and customary use in describing the
    origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
    agreed to in writing, Licensor provides the Work (and each
    Contributor provides its Contributions) on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied, including, without limitation, any warranties or conditions
    of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
    PARTICULAR PURPOSE. You are solely responsible for determining the
    appropriateness of using or redistributing the Work and assume any
    risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
    whether in tort (including negligence), contract, or otherwise,
    unless required by applicable law (such as deliberate and grossly
    negligent acts) or agreed to in writing, shall any Contributor be
    liable to You for damages, including any direct, indirect, special,
    incidental, or consequential damages of any character arising as a
    result of this License or out of the use or inability to use the
    Work (including but not limited to damages for loss of goodwill,
    work stoppage, computer failure or malfunction, or any and all
```

(continues on next page)

(continued from previous page)

```
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright 2021 Mechanics of Microstructures Group
   at The University of Manchester

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

3.5 API Documentation

Information on specific functions, classes, and methods.

3.5.1 defdap.base module

class defdap.base.**Map**
Bases: object

Base class for a map. Contains common functionality for all maps.

grainList
List of grains.

Type list of defdap.base.Grain

currGrainId
ID of last selected grain.

Type int

property xDim

property yDim

crop(*map_data*)

checkGrainsDetected(*raiseExc=True*)
Check if grains have been detected.

Parameters **raiseExc** (*bool*) – If True then an expception is raised if grains have not been detected.

Returns True if grains detected, False otherwise.

Return type bool

Raises **Exception** – If grains not detected.

plotGrainNumbers(*dilateBoundaries=False*, *ax=None*, ***kwargs*)
Plot a map with grains numbered.

Parameters

- **dilateBoundaries** (*bool, optional*) – Set to true to dilate boundaries.
- **ax** (*matplotlib.axes.Axes, optional*) – axis to plot on, if not provided the current active axis is used.
- **kwargs** (*dict, optional*) – Keyword arguments passed to *defdap.plotting.MapPlot.addGrainNumbers()*

Returns

Return type *defdap.plotting.MapPlot*

locateGrainID(*clickEvent=None*, *displaySelected=False*, ***kwargs*)
Interactive plot for identifying grains.

Parameters

- **clickEvent** (*optional*) – Click handler to use.
- **displaySelected** (*bool, optional*) – If true, plot slip traces for grain selected by click.
- **kwargs** (*dict, optional*) – Keyword arguments passed to `defdap.base.Map.plotDefault()`

clickGrainID(*event*, *plot*, *displaySelected*)
Event handler to capture clicking on a map.

Parameters

- **event** – Click event.
- **plot** (defdap.plotting.MapPlot) – Plot to capture clicks from.
- **displaySelected** (*bool*) – If true, plot the selected grain alone in pop-out window.

drawLineProfile(***kwargs*)
Interactive plot for drawing a line profile of data.

Parameters **kwargs** (*dict, optional*) – Keyword arguments passed to defdap.base.Map.plotDefault()

calcLineProfile(*plot*, *startEnd*, ***kwargs*)
Calculate and plot the line profile.

Parameters

- **plot** (defdap.plotting.MapPlot) – Plot to calculate the line profile for.
- **startEnd** (*array_like*) – Selected points (x0, y0, x1, y1).
- **kwargs** (*dict, optional*) – Keyword arguments passed to matplotlib.pyplot.plot()

setHomogPoint(*binSize=1*, *points=None*, ***kwargs*)
Interactive tool to set homologous points. Right-click on a point then click 'save point' to append to the homologous points list.

Parameters

- **binSize** (*int, optional*) – Binning applied to image, if applicable.
- **points** (*numpy.ndarray, optional*) – Array of (x,y) homologous points to set explicitly.
- **kwargs** (*dict, optional*) – Keyword arguments passed to defdap.base.Map.plotHomog()

clickHomog(*event*, *plot*)
Event handler for capturing position when clicking on a map.

Parameters

- **event** – Click event.
- **plot** (defdap.plotting.MapPlot) – Plot to monitor.

keyHomog(*event*, *plot*)
Event handler for moving position using keyboard after clicking on a map.

Parameters

- **event** – Keypress event.
- **plot** (defdap.plotting.MapPlot) – Plot to monitor.

clickSaveHomog(*event*, *plot*, *binSize*)
Append the selected point on the map to homogPoints.

Parameters

- **event** – Button click event.
- **plot** (defdap.plotting.MapPlot) – Plot to monitor.
- **binSize** (*int, optional*) – Binning applied to image, if applicable.

updateHomogPoint(*homogID*, *newPoint=None*, *delta=None*)
Update a homog point by either over writing it with a new point or incrementing the current values.

Parameters

- **homogID** (*int*) – ID (place in list) of point to update or -1 for all.
- **newPoint** (*tuple, optional*) – (x, y) coordinates of new point.
- **delta** (*tuple, optional*) – Increments to current point (dx, dy).

buildNeighbourNetwork()
Construct a list of neighbours

displayNeighbours(***kwargs*)

clickGrainNeighbours(*event*, *plot*)
Event handler to capture clicking and show neighbours of selected grain.

Parameters

- **event** – Click event.
- **plot** (defdap.plotting.MapPlot) – Plot to monitor.

property proxigram
Proxigram for a map.

Returns Distance from a grain boundary at each point in map.

Return type numpy.ndarray

calcProxigram(*numTrials=500*, *forceCalc=True*)
Calculate distance from a grain boundary at each point in map.

Parameters

- **numTrials** (*int, optional*) – number of trials.
- **forceCalc** (*bool, optional*) – Force calculation even is proxigramArr is populated.

plot_map(*map_name*, *comp=None*, ***kwargs*)
Plot a map from the DIC data.

Parameters

- **map_name** (*str*) – Map component to plot i.e. e11, f11, max_shear.
- **comp** (*tuple*) –
- **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns Plot containing map.

Return type *defdap.plotting.MapPlot*

calcGrainAv(*mapData*, *grainIds=- 1*)
Calculate grain average of any DIC map data.

Parameters

- **mapData** (*numpy.ndarray*) – Array of map data to grain average. This must be cropped!
- **grainIds** (*list, optional*) – grainIDs to perform operation on, set to -1 for all grains.

Returns Array containing the grain average values.

Return type numpy.ndarray

grainDataToMapData(*grainData*, *grainIds=- 1*, *bg=0*)
Create a map array with each grain filled with the given values.

Parameters

- **grainData** (*list or numpy.ndarray*) – Grain values. This can be a single value per grain or RGB values.
- **grainIds** (*list of int or int, optional*) – IDs of grains to plot for. Use -1 for all grains in the map.
- **bg** (*int or real, optional*) – Value to fill the background with.

Returns **grainMap** – Array filled with grain data values

Return type numpy.ndarray

plotGrainDataMap(*mapData=None*, *grainData=None*, *grainIds=- 1*, *bg=0*, ***kwargs*)
Plot a grain map with grains coloured by given data. The data can be provided as a list of values per grain or as a map which a grain average will be applied.

Parameters

- **mapData** (*numpy.ndarray, optional*) – Array of map data. This must be cropped! Either mapData or grainData must be supplied.
- **grainData** (*list or np.array, optional*) – Grain values. This an be a single value per grain or RGB values. You must supply either mapData or grainData.
- **grainIds** (*list of int or int, optional*) – IDs of grains to plot for. Use -1 for all grains in the map.
- **bg** (*int or real, optional*) – Value to fill the background with.
- **kwargs** (*dict, optional*) – Keyword arguments passed to *defdap.plotting.MapPlot.create()*

Returns **plot** – Plot object created

Return type *defdap.plotting.MapPlot*

plotGrainDataIPF(*direction*, *mapData=None*, *grainData=None*, *grainIds=- 1*, ***kwargs*)
Plot IPF of grain reference (average) orientations with points coloured by grain average values from map data.

Parameters

- **direction** (*numpy.ndarray*) – Vector of reference direction for the IPF.
- **mapData** (*numpy.ndarray*) – Array of map data. This must be cropped! Either mapData or grainData must be supplied.
- **grainData** (*list or np.array, optional*) – Grain values. This an be a single value per grain or RGB values. You must supply either mapData or grainData.
- **grainIds** (*list of int or int, optional*) – IDs of grains to plot for. Use -1 for all grains in the map.
- **kwargs** (*dict, optional*) – Keyword arguments passed to *defdap.quat.Quat.plotIPF()*

class defdap.base.**Grain**(*grainID*, *ownerMap*)

Bases: object

Base class for a grain.

grainID

Type int

ownerMap

Type *defdap.base.Map*

coordList

Type list of tuples

property extremeCoords

Coordinates of the bounding box for a grain.

Returns minimum x, minimum y, maximum x, maximum y.

Return type int, int, int, int

centreCoords(*centreType='box'*, *grainCoords=True*)

Calculates the centre of the grain, either as the centre of the bounding box or the grains centre of mass.

Parameters

- **centreType** (*str, optional, {'box', 'com'}*) – Set how to calculate the centre. Either 'box' for centre of bounding box or 'com' for centre of mass. Default is 'box'.
- **grainCoords** (*bool, optional*) – If set True the centre is returned in the grain coordinates otherwise in the map coordinates. Defaults is grain.

Returns Coordinates of centre of grain.

Return type int, int

grainOutline(*bg=nan*, *fg=0*)

Generate an array of the grain outline.

Parameters

- **bg** (*int*) – Value for points not within grain.
- **fg** (*int*) – Value for points within grain.

Returns Bounding box for grain with nan outside the grain and given number within.

Return type numpy.ndarray

plotOutline(*ax=None*, *plotScaleBar=False*, ***kwargs*)

Plot the outline of the grain.

Parameters

- **ax** (*matplotlib.axes.Axes*) – axis to plot on, if not provided the current active axis is used.
- **plotScaleBar** (*bool*) – plots the scale bar on the grain if true.
- **kwargs** (*dict*) – keyword arguments passed to *defdap.plotting.GrainPlot.addMap()*

Returns

Return type *defdap.plotting.GrainPlot*

grainData(*mapData*)
Extract this grains data from the given map data.

Parameters **mapData** (*numpy.ndarray*) – Array of map data. This must be cropped!

Returns Array containing this grains values from the given map data.

Return type numpy.ndarray

grainMapData(*mapData=None*, *grainData=None*, *bg=nan*)
Extract a single grain map from the given map data.

Parameters

- **mapData** (*numpy.ndarray*) – Array of map data. This must be cropped! Either this or 'grainData' must be supplied and 'grainData' takes precedence.
- **grainData** (*numpy.ndarray*) – Array of data at each point in the grain. Either this or 'mapData' must be supplied and 'grainData' takes precedence.
- **bg** (*various, optional*) – Value to fill the background with. Must be same dtype as input array.

Returns Grain map extracted from given data.

Return type numpy.ndarray

grainMapDataCoarse(*mapData=None*, *grainData=None*, *kernelSize=2*, *bg=nan*)
Create a coarsed data map of this grain only from the given map data. Data is coarsened using a kernel at each pixel in the grain using only data in this grain.

Parameters

- **mapData** (*numpy.ndarray*) – Array of map data. This must be cropped! Either this or 'grainData' must be supplied and 'grainData' takes precedence.
- **grainData** (*numpy.ndarray*) – List of data at each point in the grain. Either this or 'mapData' must be supplied and 'grainData' takes precedence.
- **kernelSize** (*int, optional*) – Size of kernel as the number of pixels to dilate by i.e 1 gives a 3x3 kernel.
- **bg** (*various, optional*) – Value to fill the background with. Must be same dtype as input array.

Returns Map of this grains coarsened data.

Return type numpy.ndarray

plotGrainData(*mapData=None*, *grainData=None*, ***kwargs*)
Plot a map of this grain from the given map data.

Parameters

- **mapData** (*numpy.ndarray*) – Array of map data. This must be cropped! Either this or 'grainData' must be supplied and 'grainData' takes precedence.
- **grainData** (*numpy.ndarray*) – List of data at each point in the grain. Either this or 'mapData' must be supplied and 'grainData' takes precedence.
- **kwargs** (*dict, optional*) – Keyword arguments passed to *defdap.plotting.GrainPlot.create()*

3.5.2 defdap.crystal module

class defdap.crystal.**Phase**(*name*, *laueGroup*, *spaceGroup*, *latticeParams*)
Bases: object

property cOverA

printSlipSystems()
Print a list of slip planes (with colours) and slip directions.

class defdap.crystal.**CrystalStructure**(*name*, *symmetries*, *vertices*, *faces*)
Bases: object

static lMatrix(*a*, *b*, *c*, *alpha*, *beta*, *gamma*, *convention=None*)
Construct L matrix based on Page 22 of Randle and Engle - Introduction to texture analysis

static qMatrix(*lMatrix*)
Construct matrix of reciprocal lattice vectors to transform plane normals See C. T. Young and J. L. Lytton, J. Appl. Phys., vol. 43, no. 4, pp. 1408–1417, 1972.

class defdap.crystal.**SlipSystem**(*slipPlane*, *slipDir*, *crystalStructure*, *cOverA=None*)
Bases: object

Class used for defining and performing operations on a slip system.

property slipPlaneLabel
Return the slip plane label. For example '(111)'.

Returns Slip plane label.

Return type str

property slipDirLabel
Returns the slip direction label. For example '[110]'.

Returns Slip direction label.

Return type str

generateFamily()
Generate the family of slip systems which this system belongs to.

Returns The family of slip systems.

Return type list of SlipSystem

static load(*name*, *crystalStructure*, *cOverA=None*, *groupBy='plane'*)
Load in slip systems from file. 3 integers for slip plane normal and 3 for slip direction. Returns a list of list of slip systems grouped by slip plane.

Parameters

- **name** (*str*) – Name of the slip system file (without file extension) stored in the defdap install dir or path to a file.
- **crystalStructure** (defdap.crystal.CrystalStructure) – Crystal structure of the slip systems.
- **cOverA** (*float, optional*) – C over a ratio for hexagonal crystals.
- **groupBy** (*str, optional*) – How to group the slip systems, either by slip plane ('plane') or slip system family ('family') or don't group (None).

Returns A list of list of slip systems grouped slip plane.

Return type list of list of SlipSystem

Raises **IOError** – Raised if not 6/8 integers per line.

static group(*slipSystems*, *groupBy*)
Groups slip systems by their slip plane.

Parameters

- **slipSystems** (*list of SlipSystem*) – A list of slip systems.
- **groupBy** (*str*) – How to group the slip systems, either by slip plane ('plane') or slip system family ('family').

Returns A list of list of grouped slip systems.

Return type list of list of SlipSystem

static printSlipSystemDirectory()
Prints the location where slip system definition files are stored.

defdap.crystal.**convertIdc**(*inType*, *, *dir=None*, *plane=None*)
Convert between Miller and Miller-Bravais indices.

Parameters

- **inType** (*str {'m', 'mb'}*) – Type of indices provided. If 'm' converts from Miller to Miller-Bravais, opposite for 'mb'.
- **dir** (*tuple of int or equivalent, optional*) – Direction to convert. This OR *plane* must me provided.
- **plane** (*tuple of int or equivalent, optional*) – Plane to convert. This OR *direction* must me provided.

Returns The converted plane or direction.

Return type tuple of int

defdap.crystal.**posIdc**(*vec*)
Return a consistent positive version of a set of indices.

Parameters **vec** (*tuple of int or equivalent*) – Indices to convert.

Returns Positive version of indices.

Return type tuple of int

defdap.crystal.**reduceIdc**(*vec*)
Reduce indices to lowest integers

Parameters **vec** (*tuple of int or equivalent*) – Indices to reduce.

Returns The reduced indices.

Return type tuple of int

defdap.crystal.**safeIntCast**(*vec*, *tol=0.001*)
Cast a tuple of floats to integers, raising an error if rounding is over a tolerance.

Parameters

- **vec** (*tuple of float or equivalent*) – Vector to cast.
- **tol** (*float*) – Tolerance above which an error is raised.

Returns

Return type tuple of int

Raises **ValueError** – If the rounding is over the tolerance for any value.

defdap.crystal.**strIdx**(*idx*)
String representation of an index with overbars.

Parameters **idx** (*int*) –

Returns

Return type str

3.5.3 defdap.ebsd module

class defdap.ebsd.**Map**(*fileName*, *dataType=None*)
Bases: *defdap.base.Map*

Class to encapsulate an EBSD map and useful analysis and plotting methods.

step_size
Step size in micron.

Type float

phases
List of phases.

Type list of defdap.crystal.Phase

boundaries
Map of boundaries. -1 for a boundary, 0 otherwise.

Type numpy.ndarray

phaseBoundaries
Map of phase boundaries. -1 for boundary, 0 otherwise.

Type numpy.ndarray

grains
Map of grains. Grain numbers start at 1 here but everywhere else grainID starts at 0. Regions that are smaller than the minimum grain size are given value -2. Remnant boundary points are -1.

Type numpy.ndarray

misOri
Map of misorientation.

Type numpy.ndarray

misOriAxis
Map of misorientation axis components.

Type list of numpy.ndarray

origin
Map origin (x, y). Used by linker class where origin is a homologue point of the maps.

Type tuple(int)

data

Must contain after loading data (maps):

phase [numpy.ndarray] 1-based, 0 is non-indexed points

euler_angle [numpy.ndarray] stored as (3, yDim, xDim) in radians

Derived data:

orientation [numpy.ndarray of defdap.quat.Quat] Quaterion for each point of map. Shape (yDim, xDim).

KAM [numpy.ndarray] Kernal average misorientaion map.

GND [numpy.ndarray] GND scalar map.

Nye_tensor [numpy.ndarray] 3x3 Nye tensor at each point.

Type *defdap.utils.Datastore*

loadData (*fileName*, *dataType=None*)
Load in EBSD data from file.

Parameters

- **fileName** (*str*) – Path to EBSD file, including name, excluding extension.
- **dataType** (*str, {'OxfordBinary', 'OxfordText'}*) – Format of EBSD data file.

save (*file_name*, *data_type=None*, *file_dir=''*)
Save EBSD map to file.

Parameters

- **file_name** (*str*) – Name of file to save to, it must not already exist.
- **data_type** (*str, {'OxfordText'}*) – Format of EBSD data file to save.
- **file_dir** (*str*) – Directory to save the file to.

property crystalSym
Crystal symmetry of the primary phase.

Returns Crystal symmetry

Return type str

property cOverA
C over A ratio of the primary phase

Returns C over A ratio if hexagonal crystal structure otherwise None

Return type float or None

property numPhases

property primaryPhase
Primary phase of the EBSD map.

Returns Primary phase

Return type *defdap.crystal.Phase*

property scale

plotBandContrastMap (***kwargs*)
Plot band contrast map

kwargs All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

plotEulerMap(*phases=None*, ***kwargs*)
Plot an orientation map in Euler colouring

Parameters

- **phases** (*list of int*) – Which phases to plot for
- **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

plotIPFMap(*direction*, *backgroundColour=None*, *phases=None*, ***kwargs*)
Plot a map with points coloured in IPF colouring, with respect to a given sample direction.

Parameters

- **direction** (*np.array len 3*) – Sample directiom.
- **backgroundColour** (*np.array len 3*) – Colour of background (i.e. for phases not plotted).
- **phases** (*list of int*) – Which phases to plot IPF data for.
- **kwargs** – Other arguments passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

plotPhaseMap(***kwargs*)
Plot a phase map.

Parameters **kwargs** – All arguments passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

calc_kam()
Calculates Kernel Average Misorientaion (KAM) for the EBSD map, based on a 3x3 kernel. Crystal symmetric equivalences are not considered. Stores result as *KAM*.

plotKamMap(***kwargs*)
Plot Kernel Average Misorientaion (KAM) for the EBSD map.

Parameters **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

calc_nye()
Calculates Nye tensor and related GND density for the EBSD map. Stores result as *Nye_tensor* and *GND*. Uses the crystal symmetry of the primary phase.

plotGNDMap(***kwargs*)
Plots a map of geometrically necessary dislocation (GND) density

Parameters **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

checkDataLoaded()
Checks if EBSD data is loaded

Returns True if data loaded

Return type bool

buildQuatArray()
Build quaternion array

filterData(*misOriTol=5*)

findBoundaries(*boundDef=10*)
Find grain and phase boundaries

Parameters **boundDef** (*float*) – Critical misorientation.

buildNeighbourNetwork()
Construct a list of neighbours

plotPhaseBoundaryMap(*dilate=False*, ***kwargs*)
Plot phase boundary map.

Parameters

- **dilate** (*bool*) – If true, dilate boundary.
- **kwargs** – All other arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

plotBoundaryMap(***kwargs*)
Plot grain boundary map.

Parameters **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

findGrains(*minGrainSize=10*)
Find grains and assign IDs.

Parameters **minGrainSize** (*int*) – Minimum grain area in pixels.

plotGrainMap(***kwargs*)
Plot a map with grains coloured.

Parameters **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

floodFill(*x*, *y*, *grainIndex*, *points_left*)
Flood fill algorithm that uses the x and y boundary arrays to fill a connected area around the seed point. The points are inserted into a grain object and the grain map array is updated.

Parameters

- **x** (*int*) – Seed point x for flood fill
- **y** (*int*) – Seed point y for flood fill
- **grainIndex** (*int*) – Value to fill in grain map
- **points_left** (*numpy.ndarray*) – Boolean map of the points that have not been assigned a grain yet

Returns **currentGrain** – New grain object with points added

Return type *defdap.ebsd.Grain*

calcGrainAvOris()
Calculate the average orientation of grains.

calcGrainMisOri(*calcAxis=False*)
Calculate the misorientation within grains.

Parameters **calcAxis** (*bool*) – Calculate the misorientation axis if True.

plotMisOriMap(*component=0*, ***kwargs*)
Plot misorientation map.

Parameters

- **component** (*int, {0, 1, 2, 3}*) – 0 gives misorientation, 1, 2, 3 gives rotation about x, y, z
- **kwargs** – All other arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

calcAverageGrainSchmidFactors(*loadVector*, *slipSystems=None*)
Calculates Schmid factors for all slip systems, for all grains, based on average grain orientation.

Parameters

- **loadVector** – Loading vector, e.g. [1, 0, 0].
- **slipSystems** (*list, optional*) – Slip planes to calculate Schmid factor for, maximum of all planes calculated if not given.

plotAverageGrainSchmidFactorsMap(*planes=None*, *directions=None*, ***kwargs*)
Plot maximum Schmid factor map, based on average grain orientation (for all slip systems unless specified).

Parameters

- **planes** (*list, optional*) – Plane ID(s) to consider. All planes considered if not given.
- **directions** (*list, optional*) – Direction ID(s) to consider. All directions considered if not given.
- **kwargs** – All other arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

class defdap.ebsd.**Grain**(*grainID*, *ebsdMap*)
Bases: *defdap.base.Grain*

Class to encapsulate a grain in an EBSD map and useful analysis and plotting methods.

ebsdMap
EBSD map this grain is a member of.

Type *defdap.ebsd.Map*

ownerMap
EBSD map this grain is a member of.

Type *defdap.ebsd.Map*

phaseID

Type int

phase

Type *defdap.crystal.Phase*

quatList
List of quats.

Type list

misOriList
MisOri at each point in grain.

Type list

misOriAxisList
MisOri axes at each point in grain.

Type list

refOri
Average ori of grain

Type *defdap.quat.Quat*

averageMisOri
Average misOri of grain.

averageSchmidFactors
List of list Schmid factors (grouped by slip plane).

Type list

slipTraceAngles
Slip trace angles in screen plane.

Type list

slipTraceInclinations
Angle between slip plane and screen plane.

Type list

property plotDefault

property crystalSym
Temporary

addPoint(*coord*, *quat*)
Append a coordinate and a quat to a grain.

Parameters

- **coord** (*tuple*) – (x,y) coordinate to append
- **quat** (defdap.quat.Quat) – Quaternion to append.

calcAverageOri()
Calculate the average orientation of a grain.

buildMisOriList(*calcAxis=False*)
Calculate the misorientation within given grain.

Parameters **calcAxis** (*bool*) – Calculate the misorientation axis if True.

plotRefOri(*direction=array([0, 0, 1])*, ***kwargs*)
Plot the average grain orientation on an IPF.

Parameters

- **direction** (*numpy.ndarray*) – Sample direction for IPF.
- **kwargs** – All other arguments are passed to *defdap.quat.Quat.plotIPF()*.

Returns

Return type *defdap.plotting.PolePlot*

plotOriSpread(*direction=array([0, 0, 1])*, ***kwargs*)
Plot all orientations within a given grain, on an IPF.

Parameters

- **direction** (*numpy.ndarray*) – Sample direction for IPF.
- **kwargs** – All other arguments are passed to *defdap.quat.Quat.plotIPF()*.

Returns

Return type *defdap.plotting.PolePlot*

plotUnitCell(*fig=None*, *ax=None*, *plot=None*, ***kwargs*)
Plot an unit cell of the average grain orientation.

Parameters

- **fig** (*matplotlib.figure.Figure*) – Matplotlib figure to plot on
- **ax** (*matplotlib.figure.Figure*) – Matplotlib figure to plot on
- **plot** (defdap.plotting.PolePlot) – defdap plot to plot the figure to.
- **kwargs** – All other arguments are passed to *defdap.quat.Quat.plotUnitCell()*.

plotMisOri(*component=0*, ***kwargs*)
Plot misorientation map for a given grain.

Parameters

- **component** (*int, {0, 1, 2, 3}*) – 0 gives misorientation, 1, 2, 3 gives rotation about x, y, z.
- **kwargs** – All other arguments are passed to `defdap.ebsd.plotGrainData()`.

Returns

Return type *defdap.plotting.GrainPlot*

calcAverageSchmidFactors(*loadVector*, *slipSystems=None*)
Calculate Schmid factors for grain, using average orientation.

Parameters

- **loadVector** (*numpy.ndarray*) – Loading vector, i.e. [1, 0, 0]
- **slipSystems** (*list, optional*) – Slip planes to calculate Schmid factor for. Maximum for all planes used if not set.

property slipTraces
Returns list of slip trace angles.

Returns Slip trace angles based on grain orientation in calcSlipTraces.

Return type list

printSlipTraces()
Print a list of slip planes (with colours) and slip directions

calcSlipTraces(*slipSystems=None*)
Calculates list of slip trace angles based on grain orientation.

Parameters **slipSystems** (defdap.crystal.SlipSystem, *optional*) –

class defdap.ebsd.**BoundarySegment**(*ebsdMap*, *grain1*, *grain2*)
Bases: object

addBoundaryPoint(*point*, *kind*, *ownerGrain*)

boundaryPointPairs(*kind*)
Return pairs of points either side of the boundary. The first point is always in grain1

property boundaryPointPairsX
Return pairs of points either side of the boundary. The first point is always in grain1

property boundaryPointPairsY
Return pairs of points either side of the boundary. The first point is always in grain1

property boundaryLines
Return line points along this boundary segment

misorientation()

static boundary_points_to_lines(*, *boundary_points_x=None*, *boundary_points_y=None*)

class defdap.ebsd.**Linker**(*ebsd_maps*)
Bases: object

Class for linking multiple EBSD maps of the same region for analysis of deformation.

ebsd_maps
List of *ebsd.Map* objects that are linked.

Type list(*ebsd.Map*)

links
List of grain link. Each link is stored as a tuple of grain IDs (one from each map stored in same order of maps).

Type list(tuple(int))

plots
List of last opened plot of each map.

Type list(*plotting.MapPlot*)

set_origin(***kwargs*)
Interacive tool to set origin of each EBSD map.

Parameters **kwargs** – Keyword arguments passed to `defdap.ebsd.Map.plotDefault()`

click_set_origin(*event*, *plot*)
Event handler for clicking to set origin of map.

Parameters

- **event** – Click event.
- **plot** (`defdap.plotting.MapPlot`) – Plot to capture clicks from.

start_linking()
Start interactive grain linking process of each EBSD map.

click_grain_guess(*event*, *plot*)
Guesses grain position in other maps, given click on one.

Parameters

- **event** – Click handler.
- **plot** (`defdap.plotting.Plot`) – Plot to capture clicks from.

make_link(*event*, *plot*)
Make a link between the EBSD maps after clicking.

reset_links()
Reset links.

set_ref_ori_from_master()
Loop over each map (not first/reference) and each link. Sets refOri of linked grains to refOri of grain in first map.

update_misori(*calc_axis=False*)
Recalculate misorientation for linked grain (not for first map)

Parameters **calc_axis** (*bool*) – Calculate the misorientation axis if True.

3.5.4 defdap.file_readers module

class defdap.file_readers.**EBSDDataLoader**
Bases: `object`

Class containing methods for loading and checking EBSD data

static getLoader(*dataType*)

Return type `Type[EBSDDataLoader]`

checkMetadata()
Checks that the number of phases from metadata matches the amount of phases loaded.

Return type `None`

checkData()

Return type `None`

class defdap.file_readers.**OxfordTextLoader**
Bases: *defdap.file_readers.EBSDDataLoader*

load(*fileName*, *fileDir=''*)
Read an Oxford Instruments .ctf file, which is a HKL single orientation file.

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Return type None

class defdap.file_readers.**EdaxAngLoader**
Bases: *defdap.file_readers.EBSDDataLoader*

load(*file_name*, *file_dir=''*)
Read an EDAX .ang file.

Parameters

- **file_name** (str) – File name.
- **file_dir** (str) – Path to file.

Return type None

static parse_phase(*lines*)

Return type *Phase*

class defdap.file_readers.**OxfordBinaryLoader**
Bases: *defdap.file_readers.EBSDDataLoader*

load(*fileName*, *fileDir=''*)
Read Oxford Instruments .cpr/.crc file pair.

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Return type None

loadOxfordCPR(*fileName*, *fileDir=''*)
Read an Oxford Instruments .cpr file, which is a metadata file describing EBSD data.

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Return type None

loadOxfordCRC(*fileName*, *fileDir=''*)
Read binary EBSD data from an Oxford Instruments .crc file

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Return type None

class defdap.file_readers.**PythonDictLoader**
Bases: *defdap.file_readers.EBSDDataLoader*

load(*dataDict*)
Construct EBSD data from a python dictionary.

Parameters **dataDict** (Dict[str, Any]) –

Dictionary with keys: 'step_size' 'phases' 'phase' 'euler_angle' 'band_contrast'

Return type None

class defdap.file_readers.**DICDataLoader**
Bases: object

Class containing methods for loading and checking HRDIC data

checkMetadata()

Return type None

checkData()
Calculate size of map from loaded data and check it matches values from metadata.

Return type None

loadDavisMetadata(*fileName*, *fileDir=''*)
Load DaVis metadata from Davis .txt file.

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Returns Davis metadata.

Return type dict

loadDavisData(*fileName*, *fileDir=''*)
Load displacement data from Davis .txt file containing x and y coordinates and x and y displacements for each coordinate.

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Returns Coordinates and displacements.

Return type dict

static loadDavisImageData(*fileName*, *fileDir=''*)
A .txt file from DaVis containing a 2D image

Parameters

- **fileName** (str) – File name.
- **fileDir** (str) – Path to file.

Returns Array of data.

Return type np.ndarray

defdap.file_readers.**readUntilString**(*file*, *termString*, *commentChar='*'*, *lineProcess=None*, *exact=False*)

Read lines in a file until a line starting with the *termString* is encounted. The file position is returned before the line starting with the *termString* when found. Comment and empty lines are ignored.

Parameters

- **file** (TextIO) – An open python text file object.
- **termString** (str) – String to terminate reading.
- **commentChar** (str) – Character at start of a comment line to ignore.
- **lineProcess** (Optional[Callable[[str], Any]]) – Function to apply to each line when loaded.
- **exact** (bool) – A line must exactly match *termString* to stop.

Returns List of processed lines loaded from file.

Return type list

3.5.5 defdap.file_writers module

class defdap.file_writers.**EBSDDataWriter**

Bases: object

static get_writer(*datatype*)

Return type Type[*EBSDDataLoader*]

class defdap.file_writers.**OxfordTextWriter**

Bases: *defdap.file_writers.EBSDDataWriter*

write(*file_name*, *file_dir=''*)

Write an Oxford Instruments .ctf file, which is a HKL single orientation file.

Parameters

- **file_name** (str) – File name.
- **file_dir** (str) – Path to file.

Return type None

3.5.6 defdap.hrdic module

class defdap.hrdic.**Map**(*path*, *fname*, *dataType=None*)

Bases: *defdap.base.Map*

Class to encapsulate DIC map data and useful analysis and plotting methods.

format

Software name.

Type str

version

Software version.

Type str

binning
Sub-window size in pixels.

Type int

xdim
Size of map along x (from header).

Type int

ydim
Size of map along y (from header).

Type int

shape
Size of map (after cropping, like *Dim).

Type tuple

corrVal
Correlation value.

Type numpy.ndarray

ebsdMap
EBSD map linked to DIC map.

Type *defdap.ebsd.Map*

ebsdTransform
Transform from EBSD to DIC coordinates.

Type various

ebsdTransformInv
Transform from DIC to EBSD coordinates.

Type various

ebsdGrainIds
EBSD grain IDs corresponding to DIC map grain IDs.

Type list

patternImPath
Path to BSE image of map.

Type str

plotHomog
Map to use for defining homologous points (defaults to plotMaxShear).

highlightAlpha
Alpha (transparency) of grain highlight.

Type float

bseScale
Size of a pixel in the correlated images.

Type float

patScale
Size of pixel in loaded pattern relative to pixel size of dic data i.e 1 means they are the same size and 2 means the pixels in the pattern are half the size of the dic data.

Type float

path
File path.

Type str

fname
File name.

Type str

cropDists
Crop distances (default all zeros).

Type numpy.ndarray

data

Must contain after loading data (maps):

coordinate [numpy.ndarray] X and Y coordinates

displacement [numpy.ndarray] X and Y displacements

Derived data:

f [numpy.ndarray] Components of the deformation gradient (0=x, 1=y).

e [numpy.ndarray] Components of the green strain (0=x, 1=y).

max_shear [numpy.ndarray] Max shear component np.sqrt(((e11 - e22) / 2.)**2 + e12**2).

Type *defdap.utils.Datastore*

property crystalSym

loadData(*fileDir*, *fileName*, *dataType=None*)
Load DIC data from file.

Parameters

- **fileDir** (*str*) – Path to file.
- **fileName** (*str*) – Name of file including extension.
- **dataType** (*str, {'DavisText'}*) – Type of data file.

loadCorrValData(*fileDir*, *fileName*, *dataType=None*)
Load correlation value for DIC data

Parameters

- **fileDir** (*str*) – Path to file.
- **fileName** (*str*) – Name of file including extension.
- **dataType** (*str, {'DavisImage'}*) – Type of data file.

retrieveName()
Gets the first name assigned to the a map, as a string

setScale(*micrometrePerPixel*)
Sets the scale of the map.

Parameters **micrometrePerPixel** (*float*) – Length of pixel in original BSE image in micrometres.

property scale
Returns the number of micrometers per pixel in the DIC map.

printStatsTable(*percentiles*, *components*)
Print out a statistics table for a DIC map

Parameters

- **percentiles** (*list of float*) – list of percentiles to print i.e. 0, 50, 99.
- **components** (*list of str*) – list of map components to print i.e. e11, f11, max_shear.

setCrop(*xMin=None*, *xMax=None*, *yMin=None*, *yMax=None*, *updateHomogPoints=False*)
Set a crop for the DIC map.

Parameters

- **xMin** (*int*) – Distance to crop from left in pixels.
- **xMax** (*int*) – Distance to crop from right in pixels.
- **yMin** (*int*) – Distance to crop from top in pixels.
- **yMax** (*int*) – Distance to crop from bottom in pixels.
- **updateHomogPoints** (*bool, optional*) – If true, change homologous points to reflect crop.

crop(*mapData*, *binned=True*)
Crop given data using crop parameters stored in map i.e. cropped_data = DicMap.crop(DicMap.data_to_crop).

Parameters

- **mapData** (*numpy.ndarray*) – Bap data to crop.
- **binned** (*bool*) – True if mapData is binned i.e. binned BSE pattern.

setHomogPoint(*points=None*, *display=None*, ***kwargs*)
Set homologous points. Uses interactive GUI if points is None.

Parameters

- **points** (*list, optional*) – homologous points to set.
- **display** (*string, optional*) – Use max shear map if set to 'maxshear' or pattern if set to 'pattern'.

linkEbsdMap(*ebsdMap*, *transformType='affine'*, ***kwargs*)
Calculates the transformation required to align EBSD dataset to DIC.

Parameters

- **ebsdMap** (*defdap.ebsd.Map*) – EBSD map object to link.
- **transformType** (*str, optional*) – affine, piecewiseAffine or polynomial.
- **kwargs** – All arguments are passed to *estimate* method of the transform.

checkEbsdLinked()
Check if an EBSD map has been linked.

Returns Returns True if EBSD map linked.

Return type bool

Raises **Exception** – If EBSD map not linked.

warpToDicFrame(*mapData*, *cropImage=True*, *order=1*, *preserve_range=False*)
Warps a map to the DIC frame.

Parameters

- **mapData** (*numpy.ndarray*) – Data to warp.
- **cropImage** (*bool, optional*) – Crop to size of DIC map if true.
- **order** (*int, optional*) – Order of interpolation (0: Nearest-neighbor, 1: Bi-linear...).
- **preserve_range** (*bool, optional*) – Keep the original range of values.

Returns Map (i.e. EBSD map data) warped to the DIC frame.

Return type numpy.ndarray

warp_lines_to_dic_frame(*lines*)
Warp a set of lines to the DIC reference frame.

Parameters **lines** (*list of tuples*) – Lines to warp. Each line is represented as a tuple of start and end coordinates (x, y).

Returns List of warped lines with same representation as input.

Return type list of tuples

property boundaries
Returns EBSD map grain boundaries warped to DIC frame.

property boundaryLines

property phaseBoundaryLines

generateThresholdMask(*mask*, *dilation=0*, *preview=True*)
Generate a dilated mask, based on a boolean array and previews the appication of this mask to the max shear map.

Parameters

- **mask** (*numpy.array(bool)*) – A boolean array where points to be removed are True
- **dilation** (*int, optional*) – Number of pixels to dilate the mask by. Useful to remove anomalous points around masked values. No dilation applied if not specified.
- **preview** (*bool*) – If true, show the mask and preview the masked effective shear strain map.

Examples

To remove data points in dicMap where *max_shear* is above 0.8, use:

```
>>> mask = dicMap.data.max_shear > 0.8
```

To remove data points in dicMap where e11 is above 1 or less than -1, use:

```
>>> mask = (dicMap.data.e[0, 0] > 1) | (dicMap.data.e[0, 0] < -1)
```

To remove data points in dicMap where corrVal is less than 0.4, use:

```
>>> mask = dicMap.corrVal < 0.4
```

Note: correlation value data needs to be loaded seperately from the DIC map, see `defdap.hrdic.loadCorrValData()`

applyThresholdMask()
Apply mask to all DIC map data by setting masked values to nan.

setPatternPath(*filePath*, *windowSize*)
Set the path to the image of the pattern.

Parameters

- **filePath** (*str*) – Path to image.
- **windowSize** (*float*) – Size of pixel in pattern image relative to pixel size of DIC data i.e 1 means they are the same size and 2 means the pixels in the pattern are half the size of the dic data.

plotPattern(***kwargs*)
Plot BSE image of Map. For use with setting homog points.

Parameters **kwargs** – All arguments are passed to *defdap.plotting.MapPlot.create()*.

Returns

Return type *defdap.plotting.MapPlot*

plotMaxShear(***kwargs*)
Plot a map of maximum shear strain.

Parameters **kwargs** – All arguments are passed to `defdap.hrdic.plotMap()`.

Returns Plot containing map.

Return type *defdap.plotting.MapPlot*

plotGrainAvMaxShear(***kwargs*)
Plot grain map with grains filled with average value of max shear. This uses the max shear values stored in grain objects, to plot other data use `plotGrainAv()`.

Parameters **kwargs** – All arguments are passed to *defdap.base.Map.plotGrainDataMap()*.

findGrains(*algorithm=None*, *minGrainSize=10*)
Finds grains in the DIC map.

Parameters

- **algorithm** (*str {'warp', 'floodfill'}*) – Use floodfill or warp algorithm.
- **minGrainSize** (*int*) – Minimum grain area in pixels for floodfill algorithm.

floodFill(*x*, *y*, *grainIndex*, *points_left*)
Flood fill algorithm that uses the combined x and y boundary array to fill a connected area around the seed point. The points are inserted into a grain object and the grain map array is updated.

Parameters

- **x** (*int*) – Seed point x for flood fill
- **y** (*int*) – Seed point y for flood fill
- **grainIndex** (*int*) – Value to fill in grain map
- **points_left** (*numpy.ndarray*) – Boolean map of the points that have not been assigned a grain yet

Returns **currentGrain** – New grain object with points added

Return type *defdap.hrdic.Grain*

runGrainInspector(*vmax=0.1*, *corrAngle=0*)
Run the grain inspector interactive tool.

Parameters

- **vmax** (*float*) – Maximum value of the colour map.
- **corrAngle** (*float*) – Correction angle in degrees to subtract from measured angles to account for small rotation between DIC and EBSD frames. Approximately the rotation component of affine transform.

class defdap.hrdic.**Grain**(*grainID*, *dicMap*)
Bases: *defdap.base.Grain*

Class to encapsulate DIC grain data and useful analysis and plotting methods.

dicMap
DIC map this grain is a member of

Type *defdap.hrdic.Map*

ownerMap
DIC map this grain is a member of

Type *defdap.hrdic.Map*

maxShearList
List of maximum shear values for grain.

Type list

ebsdGrain
EBSD grain ID that this DIC grain corresponds to.

Type *defdap.ebsd.Grain*

ebsdMap
EBSD map that this DIC grain belongs to.

Type *defdap.ebsd.Map*

pointsList
Start and end points for lines drawn using defdap.inspector.GrainInspector.

Type numpy.ndarray

groupsList
Groups, angles and slip systems detected for lines drawn using defdap.inspector.GrainInspector.

property plotDefault

addPoint(*coord*, *maxShear*)

plotMaxShear(***kwargs*)
Plot a maximum shear map for a grain.

Parameters **kwargs** – All arguments are passed to defdap.base.plotGrainData().

Returns

Return type *defdap.plotting.GrainPlot*

property refOri
Returns average grain orientation.

Returns

Return type *defdap.quat.Quat*

property slipTraces
Returns list of slip trace angles based on EBSD grain orientation.

Returns

Return type list

calcSlipTraces(*slipSystems=None*)
Calculates list of slip trace angles based on EBSD grain orientation.

Parameters **slipSystems** (*defdap.crystal.SlipSystem, optional*) –

calcSlipBands(*grainMapData*, *thres=None*, *min_dist=None*)
Use Radon transform to detect slip band angles.

Parameters

- **grainMapData** (*numpy.ndarray*) – Data to find bands in.
- **thres** (*float, optional*) – Normalised threshold for peaks.
- **min_dist** (*int, optional*) – Minimum angle between bands.

Returns Detected slip band angles

Return type list(float)

3.5.7 defdap.inspector module

class defdap.inspector.**GrainInspector**(*currMap*, *vmax*, *corrAngle*)
Bases: object

Class containing the interactive grain inspector tool for slip trace analysis and relative displacement ratio analysis.

draw()
Draw the main window, buttons, text boxes and axes.

gotoGrain(*event*, *plot*)
Go to a specified grain ID.

Parameters **event** (int) – Grain ID to go to.

saveLine(*event*, *plot*)
Save the start point, end point and angle of drawn line into the grain.

Parameters **event** (ndarray) – Start x, start y, end x, end y point of line passed from drawn line.

groupLines(*grain=None*)
Group the lines drawn in the current grain item using a mean shift algorithm, save the average angle and then detect the active slip planes.

groupsList is a list of line groups: [id, angle, [slip plane id], [angular deviation]

Parameters **grain** (defdap.hrdic.Grain) – Grain for which to group the slip lines.

clearAllLines(*event*, *plot*)
Clear all lines in a given grain.

removeLine(*event*, *plot*)
Remove single line [runs after submitting a text box].

Parameters **event** (int) – Line ID to remove.

redraw()
Draw items which need to be redrawn when changing grain ID.

redrawLine()
Draw items which need to be redrawn when adding a line.

runRDRGroup(*event*, *plot*)
Run RDR on a specified group, upon submitting a text box.

Parameters **event** (int) – Group ID specified from text box.

batchRunSTA(*event*, *plot*)
Run slip trace analysis on all grains which hve slip trace lines drawn.

calcRDR(*grain*, *group*, *showPlot=True*, *length=2.5*)
Calculates the relative displacement ratio for a given grain and group.

Parameters

- **grain** (int) – DIC grain ID to run RDR on.
- **group** (int) – group ID to run RDR on.
- **showPlot** (bool) – if True, show plot window.
- **length** (float) – length of perpendicular lines used for RDR.

plotRDR(*grain*, *group*, *ulist*, *vlist*, *allxlist*, *allylist*, *linRegResults*)
Plot RDR figure, including location of perpendicular lines and scatter plot of ucentered vs vcentered.

Parameters

- **grain** (int) – DIC grain to plot.
- **group** (int) – Group ID to plot.
- **ulist** (List[float]) – List of ucentered values.
- **vlist** (List[float]) – List of vcentered values.
- **allxlist** (List[float]) – List of all x values.
- **allylist** (List[float]) – List of all y values.
- **linRegResults** (LinregressResult) – Results from linear regression of ucentered vs vcentered {slope, intercept, rvalue, pvalue, stderr}.

updateFilename(*event*, *plot*)
Update class variable filename, based on text input from textbox handler.

event: Text in textbox.

saveFile(*event*, *plot*)
Save a file which contains definitions of slip lines drawn in grains [(x0, y0, x1, y1), angle, groupID] and groups of lines, defined by an average angle and identified sip plane [groupID, angle, [slip plane id(s)], [angular deviation(s)]]

loadFile(*event*, *plot*)
Load a file which contains definitions of slip lines drawn in grains [(x0, y0, x1, y1), angle, groupID] and groups of lines, defined by an average angle and identified sip plane [groupID, angle, [slip plane id(s)], [angular deviation(s)]]

3.5.8 defdap.plotting module

class defdap.plotting.**Plot**(*ax=None*, *axParams={}*, *fig=None*, *makeInteractive=False*, *title=None*, ***kwargs*)
Bases: object

Class used for creating and manipulating plots.

checkInteractive()
Checks if current plot is interactive.

Raises **Exception** – If plot is not interactive

addEventHandler(*eventName*, *eventHandler*)

addAxes(*loc*, *proj='2d'*)
Add axis to current plot

Parameters

- **loc** – Location of axis.
- **proj** (*str, {2d, 3d}*) – 2D or 3D projection.

Returns

Return type matplotlib.Axes.axes

addButton(*label*, *clickHandler*, *loc=0.8, 0.0, 0.1, 0.07*, ***kwargs*)
Add a button to the plot.

Parameters

- **label** (*str*) – Label for the button.
- **clickHandler** – Click handler to assign.
- **loc** (*list(float), len 4*) – Left, bottom, width, height.
- **kwargs** – All other arguments passed to matplotlib.widgets.Button.

addTextBox(*label*, *submitHandler=None*, *changeHandler=None*, *loc=0.8, 0.0, 0.1, 0.07*, ***kwargs*)
Add a text box to the plot.

Parameters

- **label** (*str*) – Label for the button.
- **submitHandler** – Submit handler to assign.
- **loc** (*list(float), len 4*) – Left, bottom, width, height.
- **kwargs** – All other arguments passed to matplotlib.widgets.TextBox.

Returns

Return type matplotlotlib.widgets.TextBox

addText(*ax*, *x*, *y*, *txt*, ***kwargs*)
Add text to the plot.

Parameters

- **ax** (*matplotlib.axes.Axes*) – Matplotlib axis to plot on.
- **x** (*float*) – x position.
- **y** (*float*) – y position.
- **txt** (*str*) – Text to write onto the plot.
- **kwargs** – All other arguments passed to `matplotlib.pyplot.text()`.

addArrow(*startEnd*, *persistent=False*, *clearPrev=True*, *label=None*)
Add arrow to grain plot.

Parameters

- **startEnd** (*4-tuple*) – Starting (x, y), Ending (x, y).
- **persistent** – If persistent, do not clear arrow with clearPrev.
- **clearPrev** – Clear all non-persistent arrows.
- **label** – Label to place near arrow.

setSize(*size*)
Set size of plot.

Parameters **size** (*float, float*) – Width and height in inches.

setTitle(*txt*)
Set title of plot.

Parameters **txt** (*str*) – Title to set.

lineSlice(*event*, *plot*, *action=None*)
Catch click and drag then draw an arrow.

Parameters

- **event** – Click event.
- **plot** (`defdap.plotting.Plot`) – Plot to capture clicks from.
- **action** – Further action to perform.

Examples

To use, add a click and release event handler to your plot, pointing to this function:

```
>>> plot.addEventHandler('button_press_event',lambda e, p: lineSlice(e, p))
>>> plot.addEventHandler('button_release_event', lambda e, p: lineSlice(e, p))
```

property exists

clear()
Clear plot.

draw()
Draw plot

class defdap.plotting.**MapPlot**(*callingMap*, *fig=None*, *ax=None*, *axParams={}*, *makeInteractive=False*, ***kwargs*)
Bases: *defdap.plotting.Plot*

Class for creating a map plot.

addMap(*mapData*, *vmin=None*, *vmax=None*, *cmap='viridis'*, ***kwargs*)
Add a map to a plot.

Parameters

- **mapData** (*numpy.ndarray*) – Map data to plot.
- **vmin** (*float*) – Minimum value for the colour scale.
- **vmax** (*float*) – Maximum value for the colour scale.
- **cmap** – Colour map.
- **kwargs** – Other arguments are passed to matplotlib.pyplot.imshow().

Returns

Return type matplotlib.image.AxesImage

addColourBar(*label*, *layer=0*, ***kwargs*)
Add a colour bar to plot.

Parameters

- **label** (*str*) – Label for the colour bar.
- **layer** (*int*) – Layer ID.
- **kwargs** – Other arguments are passed to matplotlib.pyplot.colorbar().

addScaleBar(*scale=None*)
Add scale bar to plot.

Parameters **scale** (*float*) – Size of a pixel in microns.

addGrainBoundaries(*kind='pixel'*, *boundaries=None*, *colour=None*, *dilate=False*, *draw=True*, ***kwargs*)
Add grain boundaries to the plot.

Parameters

- **kind** (*str, {"pixel", "line"}*) – Type of boundaries to plot, either a boundary image or a collection of line segments.
- **boundaries** (*various, optional*) – Boundaries to plot, either a boundary image or a list of pairs of coordinates representing the start and end of each boundary segment. If not provided the boundaries are loaded from the calling map.
- **colour** (*str*) – Colour of grain boundaries.
- **dilate** (*bool*) – If true, dilate the grain boundaries.
- **kind** –

Returns matplotlib.image.AxesImage if type is pixel

Return type Various

addGrainHighlights(*grainIds*, *grainColours=None*, *alpha=None*, *newLayer=False*)
Highlight grains in the plot.

Parameters

- **grainIds** (*list*) – List of grain IDs to highlight.
- **grainColours** – Colour to use for grain highlight.
- **alpha** (*float*) – Alpha (transparency) to use for grain highlight.

- **newLayer** (*bool*) – If true, make a new layer in imgLayers.

Returns

Return type matplotlib.image.AxesImage

addGrainNumbers(*fontsize=10*, ***kwargs*)
Add grain numbers to a map.

Parameters

- **fontsize** (*float*) – Font size.
- **kwargs** – Pass other arguments to `matplotlib.pyplot.text()`.

addLegend(*values*, *labels*, *layer=0*, ***kwargs*)
Add a legend to a map.

Parameters

- **values** (*list*) – Values to find colour patched for.
- **labels** (*list*) – Labels to assign to values.
- **layer** (*int*) – Image layer to generate legend for.
- **kwargs** – Pass other arguments to `matplotlib.pyplot.legend()`.

addPoints(*x*, *y*, *updateLayer=None*, ***kwargs*)
Add points to plot.

Parameters

- **x** (*float*) – x coordinate.
- **y** (*float*) – y coordinate.
- **updateLayer** (*int, optional*) – Layer to place points on
- **kwargs** – Other arguments passed to `matplotlib.pyplot.scatter()`.

classmethod create(*callingMap*, *mapData*, *fig=None*, *figParams={}*, *ax=None*, *axParams={}*, *plot=None*, *makeInteractive=False*, *plotColourBar=False*, *vmin=None*, *vmax=None*, *cmap=None*, *clabel=''*, *plotGBs=False*, *dilateBoundaries=False*, *boundaryColour=None*, *plotScaleBar=False*, *scale=None*, *highlightGrains=None*, *highlightColours=None*, *highlightAlpha=None*, ***kwargs*)
Create a plot for a map.

Parameters

- **callingMap** (base.Map) – DIC or EBSD map which called this plot.
- **mapData** (*numpy.ndarray*) – Data to be plotted.
- **fig** (*matplotlib.figure.Figure*) – Matplotlib figure to plot on.
- **figParams** – Passed to defdap.plotting.Plot.
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axis to plot on.
- **axParams** – Passed to defdap.plotting.Plot as axParams.
- **plot** (defdap.plotting.Plot) – If none, use current plot.
- **makeInteractive** – If true, make plot interactive
- **plotColourBar** (*bool*) – If true, plot a colour bar next to the map.

- **vmin** (*float, optional*) – Minimum value for the colour scale.
- **vmax** (*float, optional*) – Maximum value for the colour scale.
- **cmap** (*str*) – Colour map.
- **clabel** (*str*) – Label for the colour bar.
- **plotGBs** (*bool*) – If true, plot the grain boundaries on the map.
- **dilateBoundaries** (*bool*) – If true, dilate the grain boundaries.
- **boundaryColour** (*str*) – Colour to use for the grain boundaries.
- **plotScaleBar** (*bool*) – If true, plot a scale bar in the map.
- **scale** (*float*) – Size of pizel in microns.
- **highlightGrains** (*list(int)*) – List of grain IDs to highlight.
- **highlightColours** (*str*) – Colour to hightlight grains.
- **highlightAlpha** (*float*) – Alpha (transparency) by which to highlight grains.
- **kwargs** – All other arguments passed to *defdap.plotting.MapPlot.addMap()*

Returns

Return type *defdap.plotting.MapPlot*

class defdap.plotting.**GrainPlot**(*callingGrain*, *fig=None*, *ax=None*, *axParams={}*, *makeInteractive=False*, ***kwargs*)

Bases: *defdap.plotting.Plot*

Class for creating a map for a grain.

addMap(*mapData*, *vmin=None*, *vmax=None*, *cmap='viridis'*, ***kwargs*)

Add a map to a grain plot.

Parameters

- **mapData** (*numpy.ndarray*) – Grain data to plot
- **vmin** (*float*) – Minimum value for the colour scale.
- **vmax** (*float*) – Maximum value for the colour scale.
- **cmap** – Colour map to use.
- **kwargs** – Other arguments are passed to matplotlib.pyplot.imshow().

Returns

Return type matplotlib.image.AxesImage

addColourBar(*label*, *layer=0*, ***kwargs*)

Add colour bar to grain plot.

Parameters

- **label** (*str*) – Label to add to colour bar.
- **layer** (*int*) – Layer on which to add colourbar.
- **kwargs** – Other arguments passed to matplotlib.pyplot.colorbar().

addScaleBar(*scale=None*)

Add scale bar to grain plot.

Parameters **scale** (*float*) – Size of pixel in micron.

addTraces(*angles*, *colours*, *topOnly=False*, *pos=None*, ***kwargs*)
Add slip trace angles to grain plot. Illustrated by lines crossing through central pivot point to create a circle.

Parameters

- **angles** (*list*) – Angles of slip traces.
- **colours** (*list*) – Colours to plot.
- **topOnly** (*bool, optional*) – If true, plot only a semi-circle instead of a circle.
- **pos** (*tuple*) – Position of slip traces.
- **kwargs** – Other arguments are passed to `matplotlib.pyplot.quiver()`

addSlipTraces(*topOnly=False*, *colours=None*, *pos=None*, ***kwargs*)
Add slip traces to plot, based on the calling grain's slip systems.

Parameters

- **colours** (*list*) – Colours to plot.
- **topOnly** (*bool, optional*) – If true, plot only a semi-circle instead of a circle.
- **pos** (*tuple*) – Position of slip traces.
- **kwargs** – Other arguments are passed to `matplotlib.pyplot.quiver()`

addSlipBands(*topOnly=False*, *grainMapData=None*, *angles=None*, *pos=None*, *thres=None*, *min_dist=None*, ***kwargs*)
Add lines representing slip bands detected by Radon transform in `calcSlipBands()`.

Parameters

- **topOnly** (*bool, optional*) – If true, plot only a semi-circle instead of a circle.
- **grainMapData** – Map data to pass to *calcSlipBands()*.
- **angles** (*list(float), optional*) – List of angles to plot, otherwise, use angles detected in *calcSlipBands()*.
- **pos** (*tuple*) – Position in which to plot slip traces.
- **thres** (*float*) – Threshold to use in *calcSlipBands()*.
- **min_dist** – Minimum angle between bands in *calcSlipBands()*.
- **kwargs** – Other arguments are passed to `matplotlib.pyplot.quiver()`.

classmethod create(*callingGrain*, *mapData*, *fig=None*, *figParams={}*, *ax=None*, *axParams={}*, *plot=None*, *makeInteractive=False*, *plotColourBar=False*, *vmin=None*, *vmax=None*, *cmap=None*, *clabel=''*, *plotScaleBar=False*, *scale=None*, *plotSlipTraces=False*, *plotSlipBands=False*, ***kwargs*)
Create grain plot.

Parameters

- **callingGrain** (`base.Grain`) – DIC or EBSD grain which called this plot.
- **mapData** – Data to be plotted.
- **fig** (*matplotlib.figure.Figure*) – Matplotlib figure to plot on.
- **figParams** – Passed to defdap.plotting.Plot.
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axis to plot on.
- **axParams** – Passed to defdap.plotting.Plot as axParams.

- **plot** (defdap.plotting.Plot) – If none, use current plot.
- **makeInteractive** – If true, make plot interactive
- **plotColourBar** (*bool*) – If true, plot a colour bar next to the map.
- **vmin** (*float*) – Minimum value for the colour scale.
- **vmax** (*float*) – Maximum value for the colour scale.
- **cmap** – Colour map.
- **clabel** (*str*) – Label for the colour bar.
- **plotScaleBar** (*bool*) – If true, plot a scale bar in the map.
- **scale** (*float*) – Size of pizel in microns.
- **plotSlipTraces** (*bool*) – If true, plot slip traces with *addSlipTraces()*
- **plotSlipBands** (*bool*) – If true, plot slip traces with *addSlipBands()*
- **kwargs** – All other arguments passed to *defdap.plotting.GrainPlot.addMap()*

Returns

Return type *defdap.plotting.GrainPlot*

class defdap.plotting.**PolePlot**(*plotType*, *crystalSym*, *projection=None*, *fig=None*, *ax=None*, *axParams={}*, *makeInteractive=False*, ***kwargs*)

Bases: *defdap.plotting.Plot*

Class for creating an inverse pole figure plot.

addAxis()

Draw axes on the IPF based on crystal symmetry.

Raises **NotImplementedError** – If a crystal type other than 'cubic' or 'hexagonal' are selected.

addLine(*startPoint*, *endPoint*, *plotSyms=False*, *res=100*, ***kwargs*)

Draw lines on the IPF plot.

Parameters

- **startPoint** (*numpy.ndarray*) – Start point in crystal coordinates (i.e. [0,0,1]).
- **endPoint** (*numpy.ndarray*) – End point in crystal coordinates, (i.e. [1,0,0]).
- **plotSyms** (*bool, optional*) – If true, plot all symmetrically equivelant points.
- **res** (*int*) – Number of points within each line to plot.
- **kwargs** – All other arguments are passed to matplotlib.pyplot.plot().

labelPoint(*point*, *label*, *padX=0*, *padY=0*, ***kwargs*)

Place a label near a coordinate in the pole plot.

Parameters

- **point** (*tuple*) – (x, y) coordinate to place text.
- **label** (*str*) – Text to use in label.
- **padX** (*int, optional*) – Pad added to x coordinate.
- **padY** (*int, optional*) – Pad added to y coordinate.
- **kwargs** – Other arguments are passed to matplotlib.axes.Axes.text().

addPoints(*alphaAng*, *betaAng*, *markerColour=None*, *markerSize=None*, ***kwargs*)
Add a point to the pole plot.

Parameters

- **alphaAng** – Inclination angle to plot.
- **betaAng** – Azimuthal angle (around z axis from x in anticlockwise as per ISO) to plot.
- **markerColour** (*str or list(str), optional*) – Colour of marker. If two specified, then the point will have two semicircles of different colour.
- **markerSize** (*float*) – Size of marker.
- **kwargs** – Other arguments are passed to `matplotlib.axes.Axes.scatter()`.

Raises **Exception** – If more than two colours are specified

addColourBar(*label*, *layer=0*, ***kwargs*)
Add a colour bar to the pole plot.

Parameters

- **label** (*str*) – Label to place next to colour bar.
- **layer** (*int*) – Layer number to add the colour bar to.
- **kwargs** – Other argument are passed to `matplotlib.pyplot.colorbar()`.

addLegend(*label='Grain area (m^2)'*, *number=6*, *layer=0*, *scaling=1*, ***kwargs*)
Add a marker size legend to the pole plot.

Parameters

- **label** (*str*) – Label to place next to legend.
- **number** – Number of markers to plot in legend.
- **layer** (*int*) – Layer number to add the colour bar to.
- **scaling** (*float*) – Scaling applied to the data.
- **kwargs** – Other argument are passed to `matplotlib.pyplot.legend()`.

static stereoProject(**args*)
Stereographic projection of pole direction or pair of polar angles.

Parameters **args** (*numpy.ndarray, len 2 or 3*) – 2 arguments for polar angles or 3 arguments for pole directions.

Returns x coordinate, y coordinate

Return type float, float

Raises **Exception** – If input array has incorrect length

static lambertProject(**args*)
Lambert Projection of pole direction or pair of polar angles.

Parameters **args** (*numpy.ndarray, len 2 or 3*) – 2 arguments for polar angles or 3 arguments for pole directions.

Returns x coordinate, y coordinate

Return type float, float

Raises **Exception** – If input array has incorrect length

class defdap.plotting.**HistPlot**(*plotType='scatter'*, *axesType='linear'*, *density=True*, *fig=None*, *ax=None*, *axParams={}*, *makeInteractive=False*, ***kwargs*)

Bases: *defdap.plotting.Plot*

Class for creating a histogram.

addHist(*histData*, *bins=100*, *range=None*, *line='o'*, *label=None*, ***kwargs*)

Add a histogram to the current plot

Parameters

- **histData** (*numpy.ndarray*) – Data to be used in the histogram.
- **bins** (*int*) – Number of bins to use for histogram.
- **range** (*tuple or None, optional*) – The lower and upper range of the bins
- **line** (*str, optional*) – Marker or line type to be used.
- **label** (*str, optional*) – Label to use for data (used for legend).
- **kwargs** – Other arguments are passed to `numpy.histogram()`

addLegend(***kwargs*)

Add legend to histogram.

Parameters **kwargs** – All arguments passed to `matplotlib.axes.Axes.legend()`.

classmethod create(*histData*, *fig=None*, *figParams={}*, *ax=None*, *axParams={}*, *plot=None*, *makeInteractive=False*, *plotType='scatter'*, *axesType='linear'*, *density=True*, *bins=10*, *range=None*, *line='o'*, *label=None*, ***kwargs*)

Create a histogram plot.

Parameters

- **histData** (*numpy.ndarray*) – Data to be used in the histogram.
- **fig** (*matplotlib.figure.Figure*) – Matplotlib figure to plot on.
- **figParams** – Passed to defdap.plotting.Plot.
- **ax** (*matplotlib.axes.Axes*) – Matplotlib axis to plot on.
- **axParams** – Passed to defdap.plotting.Plot as axParams.
- **plot** (*defdap.plotting.HistPlot*) – Plot where histgram is created. If none, a new plot is created.
- **makeInteractive** (*bool, optional*) – If true, make plot interactive.
- **plotType** (*str, {'scatter', 'bar', 'barfilled', 'step'}*) – Type of plot to use
- **axesType** (*str, {'linear', 'logx', 'logy', 'loglog', 'None'}, optional*) – If 'log' is specified, logarithmic scale is used.
- **density** – If true, histogram is normalised such that the integral sums to 1.
- **bins** (*int*) – Number of bins to use for histogram.
- **range** (*tuple or None, optional*) – The lower and upper range of the bins
- **line** (*str, optional*) – Marker or line type to be used.
- **label** (*str, optional*) – Label to use for data (is used for legend).
- **kwargs** – Other arguments are passed to *defdap.plotting.HistPlot.addHist()*

Returns

Return type *defdap.plotting.HistPlot*

class defdap.plotting.**CrystalPlot**(*fig=None*, *ax=None*, *axParams={}*, *makeInteractive=False*, ***kwargs*)

Bases: *defdap.plotting.Plot*

Class for creating a 3D plot for plotting unit cells.

addVerts(*verts*, ***kwargs*)

Plots planes, defined by the vertices provided.

Parameters

- **verts** (*list*) – List of vertices.
- **kwargs** – Other arguments are passed to matplotlib.collections.PolyCollection.

3.5.9 defdap.quat module

class defdap.quat.**Quat**(**args*, *allow_southern=False*)

Bases: object

Class used to define and perform operations on quaternions. These are interpreted in the passive sense.

quatCoef

classmethod fromEulerAngles(*ph1*, *phi*, *ph2*)

Create a quat object from 3 Bunge euler angles.

Parameters

- **ph1** (float) – First Euler angle, rotation around Z in radians.
- **phi** (float) – Second Euler angle, rotation around new X in radians.
- **ph2** (float) – Third Euler angle, rotation around new Z in radians.

Returns Initialised Quat object.

Return type *defdap.quat.Quat*

classmethod fromAxisAngle(*axis*, *angle*)

Create a quat object from a rotation around an axis. This creates a quaternion to represent the passive rotation (-ve axis).

Parameters

- **axis** (ndarray) – Axis that the rotation is applied around.
- **angle** (float) – Magnitude of rotation in radians.

Returns Initialised Quat object.

Return type *defdap.quat.Quat*

eulerAngles()

Calculate the Euler angle representation for this rotation.

Returns **eulers** – Bunge euler angles (in radians).

Return type numpy.ndarray, shape 3

References

Melcher A. et al., 'Conversion of EBSD data by a quaternion based algorithm to be used for grain structure simulations', Technische Mechanik, 30(4)401 – 413

Rowenhorst D. et al., 'Consistent representations of and conversions between 3D rotations', Model. Simul. Mater. Sci. Eng., 23(8)

rotMatrix()
Calculate the rotation matrix representation for this rotation.

Returns **rotMatrix** – Rotation matrix.

Return type numpy.ndarray, shape (3, 3)

References

Melcher A. et al., 'Conversion of EBSD data by a quaternion based algorithm to be used for grain structure simulations', Technische Mechanik, 30(4)401 – 413

Rowenhorst D. et al., 'Consistent representations of and conversions between 3D rotations', Model. Simul. Mater. Sci. Eng., 23(8)

dot(*right*)
Calculate dot product between two quaternions.

Parameters **right** (*Quat*) – Right hand quaternion.

Returns Dot product.

Return type float

norm()
Calculate the norm of the quaternion.

Returns Norm of the quaternion.

Return type float

normalise()
Normalise the quaternion (turn it into an unit quaternion).

Returns Normalised quaternion.

Return type *defdap.quat.Quat*

property conjugate
Calculate the conjugate of the quaternion.

Returns Conjugate of quaternion.

Return type *defdap.quat.Quat*

transformVector(*vector*)
Transforms vector by the quaternion. For passive EBSD quaterions this is a transformation from sample space to crystal space. Perform on conjugate of quaternion for crystal to sample. For a quaternion representing a passive rotation from CS1 to CS2 and a fixed vector V defined in CS1, this gives the coordinates of V in CS2.

Parameters **vector** (*numpy.ndarray, shape 3 or equivalent*) – Vector to transform.

Returns Transformed vector.

Return type numpy.ndarray, shape 3

misOri(*right*, *symGroup*, *returnQuat=0*)

Calculate misorientation angle between 2 orientations taking into account the symmetries of the crystal structure. Angle is 2*arccos(output).

Parameters

- **right** (*Quat*) – Orientation to find misorientation to.
- **symGroup** (str) – Crystal type (cubic, hexagonal).
- **returnQuat** (Optional[int]) – What to return: 0 for minimum misorientation, 1 for symmetric equivalent with minimum misorientation, 2 for both.

Return type Tuple[float, *Quat*]

Returns

- *float* – Minimum misorientation.
- *defdap.quat.Quat* – Symmetric equivalent orientation with minimum misorientation.

misOriAxis(*right*)

Calculate misorientation axis between 2 orientations. This does not consider symmetries of the crystal structure.

Parameters **right** (defdap.quat.Quat) – Orientation to find misorientation axis to.

Returns Axis of misorientation.

Return type numpy.ndarray, shape 3

plotIPF(*direction*, *symGroup*, *projection=None*, *plot=None*, *fig=None*, *ax=None*, *plotColourBar=False*, *clabel=''*, *makeInteractive=False*, *markerColour=None*, *markerSize=40*, ***kwargs*)

Plot IPF of orientation, with relation to specified sample direction.

Parameters

- **direction** (ndarray) – Sample reference direction for IPF.
- **symGroup** (str) – Crystal type (cubic, hexagonal).
- **projection** (Optional[str]) – Projection to use. Either string (stereographic or lambert) or a function.
- **plot** (Optional[ForwardRef]) – Defdap plot to plot on.
- **fig** (Optional[ForwardRef]) – Figure to plot on, if not provided the current active axis is used.
- **ax** (Optional[ForwardRef]) – Axis to plot on, if not provided the current active axis is used.
- **makeInteractive** (Optional[bool]) – If true, make the plot interactive.
- **plotColourBar** (*bool*) – If true, plot a colour bar next to the map.
- **clabel** (*str*) – Label for the colour bar.
- **markerColour** (*str or list of str*) – Colour of markers (only used for half and half colouring, otherwise use argument c).
- **markerSize** (Optional[float]) – Size of markers (only used for half and half colouring, otherwise use argument s).

- **kwargs** – All other arguments are passed to *defdap.plotting.PolePlot.addPoints()*.

Return type *plotting.PolePlot*

plotUnitCell(*crystalStructure*, *OI=True*, *plot=None*, *fig=None*, *ax=None*, *makeInteractive=False*, ***kwargs*)

Plots a unit cell.

Parameters

- **crystalStructure** (defdap.crystal.CrystalStructure) – Crystal structure.
- **OI** (Optional[bool]) – True if using oxford instruments system.
- **plot** (Optional[ForwardRef]) – Plot object to plot to.
- **fig** (Optional[ForwardRef]) – Figure to plot on, if not provided the current active axis is used.
- **ax** (Optional[ForwardRef]) – Axis to plot on, if not provided the current active axis is used.
- **makeInteractive** (Optional[bool]) – True to make the plot interactive.
- **kwargs** – All other arguments are passed to *defdap.plotting.CrystalPlot.addVerts()*.

Return type *plotting.CrystalPlot*

static createManyQuats(*eulerArray*)

Create a an array of quats from an array of Euler angles.

Parameters **eulerArray** (ndarray) – Array of Bunge Euler angles of shape 3 x n x ... x m.

Returns **quats** – Array of quat objects of shape n x ... x m.

Return type numpy.ndarray(*defdap.quat.Quat*)

static multiplyManyQuats(*quats*, *right*)

Multiply all quats in a list of quats, by a single quat.

Parameters

- **quats** (List[*Quat*]) – List of quats to be operated on.
- **right** (*Quat*) – Single quaternion to multiply with the list of quats.

Returns

Return type list(*defdap.quat.Quat*)

static extract_quat_comps(*quats*)

Return a NumPy array of the provided quaternion components

Input quaternions may be given as a list of Quat objects or any iterable whose items have 4 components which map to the quaternion.

Parameters **quats** (*numpy.ndarray* (defdap.quat.Quat)) – A list of Quat objects to return the components of

Returns Array of quaternion components, shape (4, ..)

Return type numpy.ndarray

static calcSymEqvs(*quats*, *symGroup*, *dtype=<class 'float'>*)
Calculate all symmetrically equivalent quaternions of given quaternions.

Parameters

- **quats** (*numpy.ndarray (defdap.quat.Quat)*) – Array of quat objects.
- **symGroup** (str) – Crystal type (cubic, hexagonal).
- **dtype** (Optional[type]) – Data type used for calculation, defaults to np.float.

Returns **quatComps** – Array containing all symmetrically equivalent quaternion components of input quaternions.

Return type numpy.ndarray, shape: (numSym x 4 x numQuats)

static calcAverageOri(*quatComps*)
Calculate the average orientation of given quats.

Parameters **quatComps** (*numpy.ndarray*) – Array containing all symmetrically equivalent quaternion components of given quaternions (shape: numSym x 4 x numQuats), can be calculated with *Quat.calcSymEqvs()*.

Returns **avOri** – Average orientation of input quaternions.

Return type *defdap.quat.Quat*

static calcMisOri(*quatComps*, *refOri*)
Calculate the misorientation between the quaternions and a reference quaternion.

Parameters

- **quatComps** (ndarray) – Array containing all symmetrically equivalent quaternion components of given quaternions (shape: numSym x 4 x numQuats), can be calculated from quats with *Quat.calcSymEqvs()* .
- **refOri** (*Quat*) – Reference orientation.

Return type Tuple[ndarray, *Quat*]

Returns

- **minMisOris** (*numpy.ndarray, len numQuats*) – Minimum misorientation between quats and reference orientation.
- **minQuatComps** (*defdap.quat.Quat*) – Quaternion components describing minimum misorientation between quats and reference orientation.

static polarAngles(*x*, *y*, *z*)
Convert Cartesian coordinates to polar coordinates, for an unit vector.

Parameters

- **x** (*numpy.ndarray (float)*) – x coordinate.
- **y** (*numpy.ndarray (float)*) – y coordinate.
- **z** (*numpy.ndarray (float)*) – z coordinate.

Returns inclination angle and azimuthal angle (around z axis from x in anticlockwise as per ISO).

Return type float, float

static calcIPFcolours(*quats*, *direction*, *symGroup*, *dtype=<class 'numpy.float32'>*)
Calculate the RGB colours, based on the location of the given quats on the fundamental region of the IPF for the sample direction specified.

Parameters

- **quats** (*numpy.ndarray (defdap.quat.Quat)*) – Array of quat objects.
- **direction** (ndarray) – Direction in sample space.
- **symGroup** (str) – Crystal type (cubic, hexagonal).
- **dtype** (Optional[type]) – Data type to use for calculation.

Returns Array of rgb colours for each quat.

Return type numpy.ndarray, shape (3, numQuats)

References

Stephen Cluff (BYU) - IPF_rgbcalc.m subroutine in OpenXY https://github.com/BYU-MicrostructureOfMaterials/OpenXY/blob/master/Code/PlotIPF.m

static calcFundDirs(*quats*, *direction*, *symGroup*, *dtype=<class 'float'>*)

Transform the sample direction to crystal coords based on the quats and find the ones in the fundamental sector of the IPF.

Parameters

- **quats** (*array_like (defdap.quat.Quat)*) – Array of quat objects.
- **direction** (ndarray) – Direction in sample space.
- **symGroup** (str) – Crystal type (cubic, hexagonal).
- **dtype** (Optional[type]) – Data type to use for calculation.

Returns inclination angle and azimuthal angle (around z axis from x in anticlockwise).

Return type float, float

static symEqv(*symGroup*)

Returns all symmetric equivalents for a given crystal type. LEGACY: move to use symmetries defined in crystal structures

Parameters **symGroup** (*str*) – Crystal type (cubic, hexagonal).

Returns Symmetrically equivalent quats.

Return type list of defdap.quat.Quat

3.5.10 defdap.utils module

defdap.utils.**reportProgress**(*message=''*)

Decorator for reporting progress of given function

Parameters **message** (str) – Message to display (prefixed by 'Starting ', progress percentage and then 'Finished '

References

Inspiration from : https://gist.github.com/Garfounkel/20aa1f06234e1eedd419efe93137c004

class defdap.utils.**Datastore**
Bases: object

Storage of data and metadata, with methods to allow derived data to be calculated only when accessed.

_store
Staorage for data and metadata, keyed by data name. Each item is a dict with at least a *data* key, all other items are metadata, possibly including:

type [str]

Type of data stored: *map* - at least a 2-axis array, trailing axes are spatial

order [int] Tensor order of the data

unit [str] Measurement unit the data is stored in

plot_params [dict] Dictionary of the default paramenters used to plot

Type dict of dict

_generators
Methods to generate derived data, keyed by tuple of data names that the method produces.

Type dict

keys()
Get the names of all data items. Allows use of ***datastore* to get key-value pairs, imitating functionality of a dictionary.

add(*key*, *data*, ***kwargs*)
Add an item to the datastore.

Parameters

- **key** (*str*) – Name of the data.
- **data** (*any*) – Data to store.
- **kwargs** (*dict*) – Key-value pairs stored as the items metadata.

add_generator(*keys*, *func*, *metadatas=None*, ***kwargs*)
Add a data generator method that produces one or more data.

Parameters

- **keys** (*str or tuple of str*) – Name(s) of data that the generator produces.
- **func** (*callable*) – Method that produces the data. Should return the same number of values as there are *keys*.
- **metadatas** (*list of dict*) – Metadata dicts for each of data items produced.
- **kwargs** (*dict*) – Key-value pairs stored as the items metadata for every data item produced.

generate(*key*)
Generate data from the associated data generation method and store if metadata *save* is not set to False.

Parameters **key** (*str*) – Name of the data to generate.

Returns

Return type Requested data after generating.

update (*other*, *priority=None*)
Update with data items stored in *other*.

Parameters

- **other** (defdap.utils.Datastore) –
- **priority** (*str*) – Which datastore to keep an item from if the same name exists in both. Default is to prioritise *other*.

get_metadata (*key*, *attr*, *value=None*)
Get metadata value with a default returned if it does not exist. Imitating the *get()* method of a dictionary.

Parameters

- **key** (*str*) – Name of the data item.
- **attr** (*str*) – Metadata to get.
- **value** (*any*) – Default value to return if metadata does not exist.

Returns

Return type Metadata value or the default value.

PYTHON MODULE INDEX

d

INDEX

Symbols

A

B

E

F

G

H

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y