PolyLX Documentation

Release 0.4.9

Ondrej Lexa

Dec 15, 2017

Contents

Contents:

CHAPTER 1

Package modules

PolyLX provides following modules:

1.1 core module

Python module to visualize and analyze digitized 2D microstructures.

@author: Ondrej Lexa

Examples:

```
>>> from polylx import *
>>> g = Grains.from_shp()
>>> b = g.boundaries()
```

class polylx.core.**Boundaries**(*shapes*, *classification=None*)

Bases: polylx.core.PolySet

Class to store set of Boundaries objects

__init__(*shapes*, *classification=None*)

affine_transform(*matrix*)

Returns a transformed geometry using an affine transformation matrix. The matrix is provided as a list or tuple with 6 items: [a, b, d, e, xoff, yoff] which defines the equations for the transformed coordinates: x' = a * x + b * y + xoff y' = d * x + e * y + yoff

agg(**pairs*)

Returns concatenated result of multiple aggregations (different aggregation function for different attributes) based on actual classification. For single aggregation function use directly pandas groups, e.g. g.groups('lao', 'sao').agg(circular.mean)

Example:

```
>>> g.agg('area', np.sum, 'ead', np.mean, 'lao', circular.mean)
                area       ead        lao
class
ksp         2.443733  0.089710  76.875488
pl          1.083516  0.060629  94.197847
qtz         1.166097  0.068071  74.320337
```

barplot(*val*, ***kwargs*)
Plot seaborn swarmplot.

bootstrap(*num=100*, *size=None*)
Bootstrap random sample generator.

Args: num: number of boostraped samples. Default 100 size: size of bootstraped samples. Default number of objects.

Examples:

```
>>> bsmean = np.mean([gs.ead.mean() for gs in g.bootstrap()])
```

boundary_segments()
Create Boundaries from object boundary segments.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundary_segments()
```

boxplot(*val*, ***kwargs*)
Plot seaborn boxplot.

class_iter()

classify(**args*, ***kwargs*)
Define classification of objects.

When no aruments are provided, default unique classification based on name attribute is used.

Args:

vals: name of attribute (str) used for classification or array of values

Keywords: label: used as classification label when vals is array k: number of classes for continuous values rule: type of classification

'unique': unique value mapping (for discrete values) 'equal': k equaly spaced bins (for continuos values) 'user': bins edges defined by array k (for continuos values) 'natural': natural breaks. Default rule.

(beware not always unique solution)

'jenks': fischer jenks scheme

cmap: matplotlib colormap. Default 'viridis'

Examples:

```
>>> g.classify('name', rule='unique')
>>> g.classify('ar', rule='jenks', k=5)
```

clip(*other*)

clipstrap(*num=100*, *f=0.3*)
Bootstrap random rectangular clip generator.

Args: num: number of boostraped samples. Default 100 f: area fraction clipped from original shape. Default 0.3

Examples:

```
>>> csmean = np.mean([gs.ead.mean() for gs in g.clipstrap()])
```

countplot(***kwargs*)
Plot seaborn countplot.

df (**attrs*)
Returns `pandas.DataFrame` of object attributes.

Example:

```
>>> g.df('ead', 'ar')
```

feret (*angle=0*)
Returns array of feret diameters for given angle.

Args: angle: Caliper angle. Default 0

get (*attr*)
Returns `pandas.Series` of object attribute.

Example:

```
>>> g.get('ead')
```

get_class (*key*)

getindex (*name*)
Return the indices of the objects with given name.

gridsplit (*m=1*, *n=1*)
Rectangular split generator.

Args: m, n: number of rows and columns to split.

Examples:

```
>>> smean = np.mean([gs.ead.mean() for gs in g.gridsplit(6, 8)])
```

groups (**attrs*)
Returns `pandas.GroupBy` of object attributes.

Note that grouping is based on actual classification.

Example:

```
>>> g.classify('ar', 'natural')
>>> g.groups('ead').mean()
                       ead
class
1.01765-1.31807  0.067772
1.31807-1.5445   0.076042
1.5445-1.83304   0.065900
1.83304-2.36773  0.073338
2.36773-12.1571  0.084016
```

nndist (***kwargs*)

paror (*angles=range(0, 180)*, *normalized=True*)
Returns paror function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

plot (***kwargs*)
Plot set of `Grains` or `Boundaries` objects.

Keywords: show: If True matplotlib show is called. Default True alpha: transparency. Default 0.8 pos: legend position "top", "right" or "none". Defalt "auto" ncol: number of columns for legend. legend: Show legend. Default True show_fid: Show FID of objects. Default False show_index: Show index of objects. Default False

When show=False, returns matplotlib axes object.

proj(*angle=0*)
Returns array of cumulative projection of object for given angle. Args:

angle: angle of projection line

regularize(***kwargs*)

rose(***kwargs*)
Plot polar histogram of `Grains` or `Boundaries` orientations

Keywords: show: If True matplotlib show is called. Default True attr: property used for orientation. Default 'lao' bins: number of bins weights: if provided histogram is weighted density: True for probability density otherwise counts grid: True to show grid

When show=False, returns matplotlib axes object.

rotate(*angle*, ***kwargs*)
Returns a rotated geometry on a 2D plane. The angle of rotation can be specified in either degrees (default) or radians by setting use_radians=True. Positive angles are counter-clockwise and negative are clockwise rotations. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point.

savefig(***kwargs*)
Save grains or boudaries plot to file.

Args: filename: file to save figure. Default "figure.png" dpi: DPI of image. Default 150 See *plot* for other kwargs

scale(***kwargs*)
Returns a scaled geometry, scaled by factors along each dimension. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point. Negative scale factors will mirror or reflect coordinates.

simplify(*method='vw'*, ***kwargs*)

skew(***kwargs*)
Returns a skewed geometry, sheared by angles 'xs' along x and 'ys' along y direction. The shear angle can be specified in either degrees (default) or radians by setting use_radians=True. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or a coordinate tuple (x0, y0) for fixed point.

smooth(*method='chaikin'*, ***kwargs*)

surfor(*angles=range(0, 180)*, *normalized=True*)
Returns surfor function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

swarmplot(*val*, ***kwargs*)
Plot seaborn swarmplot.

translate(***kwargs*)
Returns a translated geometry shifted by offsets 'xoff' along x and 'yoff' along y direction.

ar
Returns array of axial ratios

Note that axial ratio is calculated from long and short axes calculated by actual `shape method`.

area
Return array of areas of the objects. For boundary returns 0.

centroid
Returns the 2D array of geometric centers of the objects

class_names

extent
Returns minimum bounding region (minx, miny, maxx, maxy) of all objects

fid
Return array of fids of objects.

height
Returns height of extent.

la
Return array of long axes of objects according to shape_method.

lao
Return array of long axes of objects according to shape_method

length
Return array of lengths of the objects.

ma
Returns mean axis

Return array of mean axes calculated by actual `shape method`.

name
Return list of names of the objects.

names
Returns list of unique object names.

representative_point
Returns a 2D array of cheaply computed points that are guaranteed to be within the objects.

sa
Return array of long axes of objects according to shape_method

sao
Return array of long axes of objects according to shape_method

shape
Return list of shapely objects.

shape_method
Set or returns shape methods of all objects.

width
Returns width of extent.

class `polylx.core.`**Boundary**(*shape*, *name='None-None'*, *fid=0*)
Bases: `polylx.core.PolyShape`

Boundary class to store polyline boundary geometry

A two-dimensional linear ring.

__init__(*shape*, *name='None-None'*, *fid=0*)
Create `Boundary` object

affine_transform(*matrix*)
Returns a transformed geometry using an affine transformation matrix. The matrix is provided as a list or tuple with 6 items: [a, b, d, e, xoff, yoff] which defines the equations for the transformed coordinates: x' = a * x + b * y + xoff y' = d * x + e * y + yoff

boundary_segments()
Create Boundaries from object boundary segments.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundaries()
>>> bs1 = g[10].boundary_segments()
>>> bs2 = b[10].boundary_segments()
```

chaikin (***kwargs*)
Chaikin corner-cutting smoothing algorithm.

Keywords: repeat: Number of repetitions. Default 2

contains (*other*)
Returns True if the geometry contains the other, else False

copy ()

cov ()
shape_method: cov

Short and long axes are calculated from eigenvalue analysis of coordinate covariance matrix.

crosses (*other*)
Returns True if the geometries cross, else False

difference (*other*)
Returns the difference of the geometries

disjoint (*other*)
Returns True if geometries are disjoint, else False

distance (*other*)
Unitless distance to other geometry (float)

dp (***kwargs*)
Douglas–Peucker simplification.

Keywords: tolerance: All points in the simplified object will be within the tolerance distance of the original geometry. Default Auto

equals (*other*)
Returns True if geometries are equal, else False

equals_exact (*other*, *tolerance*)
Returns True if geometries are equal to within a specified tolerance

feret (*angle=0*)
Returns the ferret diameter for given angle.

Args: angle: angle of caliper rotation

intersection (*other*)
Returns the intersection of the geometries

intersects (*other*)
Returns True if geometries intersect, else False

maxferet ()
shape_method: maxferet

Long axis is defined as the maximum caliper of the polyline. Short axis correspond to caliper orthogonal to long axis. Center coordinates are set to centroid of polyline.

overlaps (*other*)
Returns True if geometries overlap, else False

paror (*angles=range(0, 180)*, *normalized=True*)
Returns paror function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

plot(***kwargs*)
View `Boundary` geometry on figure.

proj(*angle=0*)
Returns the cumulative projection of object for given angle.

Args: angle: angle of projection line

regularize(***kwargs*)
Boundary vertices regularization.

Returns `Boundary` object defined by vertices regularly distributed along original `Boundary`.

Keywords: N: Number of vertices. Default 128. length: approx. length of segments. Default None

relate(*other*)
Returns the DE-9IM intersection matrix for the two geometries (string)

rotate(*angle*, ***kwargs*)
Returns a rotated geometry on a 2D plane. The angle of rotation can be specified in either degrees (default) or radians by setting use_radians=True. Positive angles are counter-clockwise and negative are clockwise rotations. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point.

scale(***kwargs*)
Returns a scaled geometry, scaled by factors along each dimension. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point. Negative scale factors will mirror or reflect coordinates.

show(***kwargs*)
Show plot of `Boundary` objects.

skew(***kwargs*)
Returns a skewed geometry, sheared by angles 'xs' along x and 'ys' along y direction. The shear angle can be specified in either degrees (default) or radians by setting use_radians=True. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or a coordinate tuple (x0, y0) for fixed point.

surfor(*angles=range(0, 180)*, *normalized=True*)
Returns surfor function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

symmetric_difference(*other*)
Returns the symmetric difference of the geometries (Shapely geometry)

touches(*other*)
Returns True if geometries touch, else False

translate(***kwargs*)
Returns a translated geometry shifted by offsets 'xoff' along x and 'yoff' along y direction.

union(*other*)
Returns the union of the geometries (Shapely geometry)

vw(***kwargs*)
Visvalingam-Whyatt simplification.

The Visvalingam-Whyatt algorithm eliminates points based on their effective area. A points effective area is defined as the change in total area of the polygon by adding or removing that point.

Keywords: threshold: Allowed total boundary length change in percents. Default 1

within(*other*)
Returns True if geometry is within the other, else False

ar
Returns axial ratio

Note that axial ratio is calculated from long and short axes calculated by actual `shape method`.

area
Area of the shape. For boundary returns 0.

bounds
Returns minimum bounding region (minx, miny, maxx, maxy)

centroid
Returns the geometric center of the object

hull
Returns array of vertices on convex hull of boundary geometry.

length
Unitless length of the geometry (float)

ma
Returns mean axis

Mean axis is calculated as square root of long axis multiplied by short axis. Both axes are calculated by actual `shape method`.

representative_point
Returns a cheaply computed point that is guaranteed to be within the object.

shape_method
Returns shape method in use

xy
Returns array of vertex coordinate pair.

class `polylx.core.`**Grain**(*shape*, *name='None'*, *fid=0*)
Bases: `polylx.core.PolyShape`

Grain class to store polygonal grain geometry

A two-dimensional grain bounded by a linear ring with non-zero area. It may have one or more negative-space "holes" which are also bounded by linear rings.

Properties: shape: `shapely.geometry.polygon.Polygon` object name: string with phase name. Default "None" fid: feature id. Default 0 shape_method: Method to calculate axes and orientation

__init__(*shape*, *name='None'*, *fid=0*)
Create `Grain` object

affine_transform(*matrix*)
Returns a transformed geometry using an affine transformation matrix. The matrix is provided as a list or tuple with 6 items: [a, b, d, e, xoff, yoff] which defines the equations for the transformed coordinates: x' = a * x + b * y + xoff y' = d * x + e * y + yoff

boundary_segments()
Create Boundaries from object boundary segments.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundaries()
>>> bs1 = g[10].boundary_segments()
>>> bs2 = b[10].boundary_segments()
```

chaikin(***kwargs*)
Chaikin corner-cutting smoothing algorithm.

Keywords: repeat: Number of repetitions. Default 2

contains(*other*)
Returns True if the geometry contains the other, else False

copy()

cov()
shape_method: cov

Short and long axes are calculated from eigenvalue analysis of coordinate covariance matrix. Center coordinates are set to centroid of exterior.

crosses(*other*)
Returns True if the geometries cross, else False

difference(*other*)
Returns the difference of the geometries

direct()
shape_method: direct

Short, long axes and centre coordinates are calculated from direct least-square ellipse fitting. If direct fitting is not possible silently fallback to moment. Center coordinates are set to centre of fitted ellipse.

disjoint(*other*)
Returns True if geometries are disjoint, else False

distance(*other*)
Unitless distance to other geometry (float)

dp(***kwargs*)
Douglas–Peucker simplification.

Keywords: tolerance: All points in the simplified object will be within the tolerance distance of the original geometry. Default Auto

equals(*other*)
Returns True if geometries are equal, else False

equals_exact(*other*, *tolerance*)
Returns True if geometries are equal to within a specified tolerance

feret(*angle=0*)
Returns the ferret diameter for given angle.

Args: angle: angle of caliper rotation

classmethod from_coords(*x*, *y*, *name='None'*, *fid=0*)
Create `Grain` from coordinate arrays

Example:

```
>>> g=Grain.from_coords([0,0,2,2],[0,1,1,0])
>>> g.xy
array([[ 0.,  0.,  2.,  2.,  0.],
       [ 0.,  1.,  1.,  0.,  0.]])
```

intersection(*other*)
Returns the intersection of the geometries

intersects(*other*)
Returns True if geometries intersect, else False

maee()
shape_method: maee

Short and long axes are calculated from minimum volume enclosing ellipse. The solver is based on Khachiyan Algorithm, and the final solution is different from the optimal value by the pre-specified amount of tolerance of EAD/100. Center coordinates are set to centre of fitted ellipse.

maxferet()
shape_method: maxferet

Long axis is defined as the maximum caliper of the polygon. Short axis correspond to caliper orthogonal to long axis. Center coordinates are set to centroid of exterior.

minbox()
shape_method: minbox

Short and long axes are claculated as widht and height of smallest area enclosing box. Center coordinates are set to centre of box.

minferet()
shape_method: minferet

Short axis is defined as the minimum caliper of the polygon. Long axis correspond to caliper orthogonal to short axis. Center coordinates are set to centroid of exterior.

moment()
shape_method: moment

Short and long axes are calculated from area moments of inertia. Center coordinates are set to centroid. If moment fitting failed silently fallback to maxferet. Center coordinates are set to centroid.

overlaps(*other*)
Returns True if geometries overlap, else False

paror(*angles=range(0, 180)*, *normalized=True*)
Returns paror function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

plot(***kwargs*)
Plot `Grain` geometry on figure.

Note that plotted ellipse reflects actual shape method

proj(*angle=0*)
Returns the cumulative projection of object for given angle.

Args: angle: angle of projection line

regularize(***kwargs*)
Grain vertices regularization.

Returns `Grain` object defined by vertices regularly distributed along boundaries of original `Grain`.

Keywords: N: Number of vertices. Default 128. length: approx. length of segments. Default None

relate(*other*)
Returns the DE-9IM intersection matrix for the two geometries (string)

rotate(*angle*, ***kwargs*)
Returns a rotated geometry on a 2D plane. The angle of rotation can be specified in either degrees (default) or radians by setting use_radians=True. Positive angles are counter-clockwise and negative are clockwise rotations. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point.

scale(***kwargs*)
Returns a scaled geometry, scaled by factors along each dimension. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point. Negative scale factors will mirror or reflect coordinates.

shape_vector(***kwargs*)
Returns shape (feature) vector.

Shape (feature) vector is calculated from Fourier descriptors (FD) to index the shape. To achieve rotation invariance, phase information of the FDs are ignored and only the magnitudes **|FDn|** are used.

Scale invariance is achieved by dividing the magnitudes by the DC component, i.e., |FD0|. Since centroid distance is a real value function, only half of the FDs are needed to index the shape.

Keywords:

N: number of points to regularize shape. Default 128 Routine return N/2 of FDs

show (***kwargs*)
Show plot of `Grain` objects.

skew (***kwargs*)
Returns a skewed geometry, sheared by angles 'xs' along x and 'ys' along y direction. The shear angle can be specified in either degrees (default) or radians by setting use_radians=True. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or a coordinate tuple (x0, y0) for fixed point.

spline (***kwargs*)
Spline based smoothing of grains.

Keywords: densify: factor for geometry densification. Default 5

surfor (*angles=range(0, 180)*, *normalized=True*)
Returns surfor function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

symmetric_difference (*other*)
Returns the symmetric difference of the geometries (Shapely geometry)

touches (*other*)
Returns True if geometries touch, else False

translate (***kwargs*)
Returns a translated geometry shifted by offsets 'xoff' along x and 'yoff' along y direction.

union (*other*)
Returns the union of the geometries (Shapely geometry)

vw (***kwargs*)
Visvalingam-Whyatt simplification.

The Visvalingam-Whyatt algorithm eliminates points based on their effective area. A points effective area is defined as the change in total area of the polygon by adding or removing that point.

Keywords: threshold: Allowed total boundary length change in percents. Default 1

within (*other*)
Returns True if geometry is within the other, else False

ar
Returns axial ratio

Note that axial ratio is calculated from long and short axes calculated by actual `shape method`.

area
Area of the shape. For boundary returns 0.

bounds
Returns minimum bounding region (minx, miny, maxx, maxy)

cdir
Returns centroid-vertex directions of grain exterior

cdist
Returns centroid-vertex distances of grain exterior

centroid
Returns the geometric center of the object

ead
Returns equal area diameter of grain

hull
Returns array of vertices on convex hull of grain geometry.

interiors
Returns list of arrays of vertex coordinate pair of interiors.

length
Unitless length of the geometry (float)

ma
Returns mean axis

Mean axis is calculated as square root of long axis multiplied by short axis. Both axes are calculated by actual `shape method`.

nholes
Returns number of holes (shape interiors)

representative_point
Returns a cheaply computed point that is guaranteed to be within the object.

shape_method
Returns shape method in use

xy
Returns array of vertex coordinate pair.

Note that only vertexes from exterior boundary are returned. For interiors use interiors property.

class `polylx.core.`**Grains**(*shapes*, *classification=None*)
Bases: `polylx.core.PolySet`

Class to store set of `Grains` objects

__init__(*shapes*, *classification=None*)

affine_transform(*matrix*)
Returns a transformed geometry using an affine transformation matrix. The matrix is provided as a list or tuple with 6 items: [a, b, d, e, xoff, yoff] which defines the equations for the transformed coordinates: x' = a * x + b * y + xoff y' = d * x + e * y + yoff

agg(**pairs*)
Returns concatenated result of multiple aggregations (different aggregation function for different attributes) based on actual classification. For single aggregation function use directly pandas groups, e.g. g.groups('lao', 'sao').agg(circular.mean)

Example:

```
>>> g.agg('area', np.sum, 'ead', np.mean, 'lao', circular.mean)
                area       ead        lao
class
ksp         2.443733  0.089710  76.875488
pl          1.083516  0.060629  94.197847
qtz         1.166097  0.068071  74.320337
```

barplot(*val*, ***kwargs*)
Plot seaborn swarmplot.

bootstrap(*num=100*, *size=None*)
Bootstrap random sample generator.

Args: num: number of boostraped samples. Default 100 size: size of bootstraped samples. Default number of objects.

Examples:

```
>>> bsmean = np.mean([gs.ead.mean() for gs in g.bootstrap()])
```

boundaries(*T=None*)
Create Boundaries from Grains.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundaries()
```

boundary_segments()
Create Boundaries from object boundary segments.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundary_segments()
```

boxplot(*val*, ***kwargs*)
Plot seaborn boxplot.

class_iter()

classify(**args*, ***kwargs*)
Define classification of objects.

When no aruments are provided, default unique classification based on name attribute is used.

Args:

vals: name of attribute (str) used for classification or array of values

Keywords: label: used as classification label when vals is array k: number of classes for continuous values rule: type of classification

'unique': unique value mapping (for discrete values) 'equal': k equaly spaced bins (for continuos values) 'user': bins edges defined by array k (for continuos values) 'natural': natural breaks. Default rule.

(beware not always unique solution)

'jenks': fischer jenks scheme

cmap: matplotlib colormap. Default 'viridis'

Examples:

```
>>> g.classify('name', rule='unique')
>>> g.classify('ar', rule='jenks', k=5)
```

clip(*other*)

clipstrap(*num=100*, *f=0.3*)
Bootstrap random rectangular clip generator.

Args: num: number of boostraped samples. Default 100 f: area fraction clipped from original shape. Default 0.3

Examples:

```
>>> csmean = np.mean([gs.ead.mean() for gs in g.clipstrap()])
```

countplot(***kwargs*)
Plot seaborn countplot.

df(**attrs*)
Returns `pandas.DataFrame` of object attributes.

Example:

```
>>> g.df('ead', 'ar')
```

feret (*angle=0*)
Returns array of feret diameters for given angle.

Args: angle: Caliper angle. Default 0

classmethod from_shp (*filename='/home/docs/checkouts/readthedocs.org/user_builds/polylx/checkouts/develop/polyl*
phasefield='phase', *phase='None'*)
Create Grains from ESRI shapefile.

Args: filename: filename of shapefile. Default sg2.shp from examples phasefield: name of attribute in shapefile that

holds names of grains or None. Default "phase".

phase: value used for grain phase when phasefield is None

get (*attr*)
Returns `pandas.Series` of object attribute.

Example:

```
>>> g.get('ead')
```

get_class (*key*)

getindex (*name*)
Return the indices of the objects with given name.

gridsplit (*m=1*, *n=1*)
Rectangular split generator.

Args: m, n: number of rows and columns to split.

Examples:

```
>>> smean = np.mean([gs.ead.mean() for gs in g.gridsplit(6, 8)])
```

groups (**attrs*)
Returns `pandas.GroupBy` of object attributes.

Note that grouping is based on actual classification.

Example:

```
>>> g.classify('ar', 'natural')
>>> g.groups('ead').mean()
                       ead
class
1.01765-1.31807  0.067772
1.31807-1.5445   0.076042
1.5445-1.83304   0.065900
1.83304-2.36773  0.073338
2.36773-12.1571  0.084016
```

nndist (***kwargs*)

paror (*angles=range(0, 180)*, *normalized=True*)
Returns paror function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

plot (***kwargs*)
Plot set of `Grains` or `Boundaries` objects.

Keywords: show: If True matplotlib show is called. Default True alpha: transparency. Default 0.8 pos: legend position "top", "right" or "none". Defalt "auto" ncol: number of columns for legend. legend: Show legend. Default True show_fid: Show FID of objects. Default False show_index: Show index of objects. Default False

When show=False, returns matplotlib axes object.

proj(*angle=0*)
Returns array of cumulative projection of object for given angle. Args:

angle: angle of projection line

regularize(***kwargs*)

rose(***kwargs*)
Plot polar histogram of `Grains` or `Boundaries` orientations

Keywords: show: If True matplotlib show is called. Default True attr: property used for orientation. Default 'lao' bins: number of bins weights: if provided histogram is weighted density: True for probability density otherwise counts grid: True to show grid

When show=False, returns matplotlib axes object.

rotate(*angle*, ***kwargs*)
Returns a rotated geometry on a 2D plane. The angle of rotation can be specified in either degrees (default) or radians by setting use_radians=True. Positive angles are counter-clockwise and negative are clockwise rotations. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point.

savefig(***kwargs*)
Save grains or boudaries plot to file.

Args: filename: file to save figure. Default "figure.png" dpi: DPI of image. Default 150 See *plot* for other kwargs

scale(***kwargs*)
Returns a scaled geometry, scaled by factors along each dimension. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point. Negative scale factors will mirror or reflect coordinates.

shape_vector(***kwargs*)
Returns array of shape (feature) vectors.

Keywords:

N: number of points to regularize shape. Default 128 Routine return N/2 of FDs

simplify(*method='vw'*, ***kwargs*)

skew(***kwargs*)
Returns a skewed geometry, sheared by angles 'xs' along x and 'ys' along y direction. The shear angle can be specified in either degrees (default) or radians by setting use_radians=True. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or a coordinate tuple (x0, y0) for fixed point.

smooth(*method='chaikin'*, ***kwargs*)

surfor(*angles=range(0, 180)*, *normalized=True*)
Returns surfor function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

swarmplot(*val*, ***kwargs*)
Plot seaborn swarmplot.

translate(***kwargs*)
Returns a translated geometry shifted by offsets 'xoff' along x and 'yoff' along y direction.

ar
Returns array of axial ratios

Note that axial ratio is calculated from long and short axes calculated by actual `shape method`.

area
Return array of areas of the objects. For boundary returns 0.

centroid
Returns the 2D array of geometric centers of the objects

class_names

ead
Returns array of equal area diameters of grains

extent
Returns minimum bounding region (minx, miny, maxx, maxy) of all objects

fid
Return array of fids of objects.

height
Returns height of extent.

la
Return array of long axes of objects according to shape_method.

lao
Return array of long axes of objects according to shape_method

length
Return array of lengths of the objects.

ma
Returns mean axis

Return array of mean axes calculated by actual `shape method`.

name
Return list of names of the objects.

names
Returns list of unique object names.

nholes
Returns array of number of holes (shape interiors)

representative_point
Returns a 2D array of cheaply computed points that are guaranteed to be within the objects.

sa
Return array of long axes of objects according to shape_method

sao
Return array of long axes of objects according to shape_method

shape
Return list of shapely objects.

shape_method
Set or returns shape methods of all objects.

width
Returns width of extent.

class polylx.core.**PolySet**(*shapes*, *classification=None*)
Bases: `object`

Base class to store set of `Grains` or `Boundaries` objects

Properties: polys: list of objects extent: tuple of (xmin, ymin, xmax, ymax)

__init__(*shapes*, *classification=None*)

affine_transform(*matrix*)
Returns a transformed geometry using an affine transformation matrix. The matrix is provided as a list or tuple with 6 items: [a, b, d, e, xoff, yoff] which defines the equations for the transformed coordinates: x' = a * x + b * y + xoff y' = d * x + e * y + yoff

agg(**pairs*)
Returns concatenated result of multiple aggregations (different aggregation function for different attributes) based on actual classification. For single aggregation function use directly pandas groups, e.g. g.groups('lao', 'sao').agg(circular.mean)

Example:

```
>>> g.agg('area', np.sum, 'ead', np.mean, 'lao', circular.mean)
                area       ead        lao
class
ksp         2.443733  0.089710  76.875488
pl          1.083516  0.060629  94.197847
qtz         1.166097  0.068071  74.320337
```

barplot(*val*, ***kwargs*)
Plot seaborn swarmplot.

bootstrap(*num=100*, *size=None*)
Bootstrap random sample generator.

Args: num: number of boostraped samples. Default 100 size: size of bootstraped samples. Default number of objects.

Examples:

```
>>> bsmean = np.mean([gs.ead.mean() for gs in g.bootstrap()])
```

boundary_segments()
Create Boundaries from object boundary segments.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundary_segments()
```

boxplot(*val*, ***kwargs*)
Plot seaborn boxplot.

class_iter()

classify(**args*, ***kwargs*)
Define classification of objects.

When no aruments are provided, default unique classification based on name attribute is used.

Args:

vals: name of attribute (str) used for classification or array of values

Keywords: label: used as classification label when vals is array k: number of classes for continuous values rule: type of classification

'unique': unique value mapping (for discrete values) 'equal': k equaly spaced bins (for continuos values) 'user': bins edges defined by array k (for continuos values) 'natural': natural breaks. Default rule.

(beware not always unique solution)

'jenks': fischer jenks scheme

cmap: matplotlib colormap. Default 'viridis'

Examples:

```
>>> g.classify('name', rule='unique')
>>> g.classify('ar', rule='jenks', k=5)
```

clip (*other*)

clipstrap (*num=100*, *f=0.3*)

Bootstrap random rectangular clip generator.

Args: num: number of boostraped samples. Default 100 f: area fraction clipped from original shape. Default 0.3

Examples:

```
>>> csmean = np.mean([gs.ead.mean() for gs in g.clipstrap()])
```

countplot (***kwargs*)

Plot seaborn countplot.

df (**attrs*)

Returns `pandas.DataFrame` of object attributes.

Example:

```
>>> g.df('ead', 'ar')
```

feret (*angle=0*)

Returns array of feret diameters for given angle.

Args: angle: Caliper angle. Default 0

get (*attr*)

Returns `pandas.Series` of object attribute.

Example:

```
>>> g.get('ead')
```

get_class (*key*)

getindex (*name*)

Return the indices of the objects with given name.

gridsplit (*m=1*, *n=1*)

Rectangular split generator.

Args: m, n: number of rows and columns to split.

Examples:

```
>>> smean = np.mean([gs.ead.mean() for gs in g.gridsplit(6, 8)])
```

groups (**attrs*)

Returns `pandas.GroupBy` of object attributes.

Note that grouping is based on actual classification.

Example:

```
>>> g.classify('ar', 'natural')
>>> g.groups('ead').mean()
                       ead
class
1.01765-1.31807   0.067772
1.31807-1.5445    0.076042
```

```
1.5445-1.83304   0.065900
1.83304-2.36773  0.073338
2.36773-12.1571  0.084016
```

nndist(***kwargs*)

paror(*angles=range(0, 180)*, *normalized=True*)

Returns paror function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

plot(***kwargs*)

Plot set of `Grains` or `Boundaries` objects.

Keywords: show: If True matplotlib show is called. Default True alpha: transparency. Default 0.8 pos: legend position "top", "right" or "none". Defalt "auto" ncol: number of columns for legend. legend: Show legend. Default True show_fid: Show FID of objects. Default False show_index: Show index of objects. Default False

When show=False, returns matplotlib axes object.

proj(*angle=0*)

Returns array of cumulative projection of object for given angle. Args:

angle: angle of projection line

regularize(***kwargs*)

rose(***kwargs*)

Plot polar histogram of `Grains` or `Boundaries` orientations

Keywords: show: If True matplotlib show is called. Default True attr: property used for orientation. Default 'lao' bins: number of bins weights: if provided histogram is weighted density: True for probability density otherwise counts grid: True to show grid

When show=False, returns matplotlib axes object.

rotate(*angle*, ***kwargs*)

Returns a rotated geometry on a 2D plane. The angle of rotation can be specified in either degrees (default) or radians by setting use_radians=True. Positive angles are counter-clockwise and negative are clockwise rotations. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point.

savefig(***kwargs*)

Save grains or boudaries plot to file.

Args: filename: file to save figure. Default "figure.png" dpi: DPI of image. Default 150 See *plot* for other kwargs

scale(***kwargs*)

Returns a scaled geometry, scaled by factors along each dimension. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point. Negative scale factors will mirror or reflect coordinates.

simplify(*method='vw'*, ***kwargs*)

skew(***kwargs*)

Returns a skewed geometry, sheared by angles 'xs' along x and 'ys' along y direction. The shear angle can be specified in either degrees (default) or radians by setting use_radians=True. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or a coordinate tuple (x0, y0) for fixed point.

smooth(*method='chaikin'*, ***kwargs*)

surfor(*angles=range(0, 180)*, *normalized=True*)

Returns surfor function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

swarmplot (*val*, ***kwargs*)
Plot seaborn swarmplot.

translate (***kwargs*)
Returns a translated geometry shifted by offsets 'xoff' along x and 'yoff' along y direction.

ar
Returns array of axial ratios

Note that axial ratio is calculated from long and short axes calculated by actual `shape method`.

area
Return array of areas of the objects. For boundary returns 0.

centroid
Returns the 2D array of geometric centers of the objects

class_names

extent
Returns minimum bounding region (minx, miny, maxx, maxy) of all objects

fid
Return array of fids of objects.

height
Returns height of extent.

la
Return array of long axes of objects according to shape_method.

lao
Return array of long axes of objects according to shape_method

length
Return array of lengths of the objects.

ma
Returns mean axis

Return array of mean axes calculated by actual `shape method`.

name
Return list of names of the objects.

names
Returns list of unique object names.

representative_point
Returns a 2D array of cheaply computed points that are guaranteed to be within the objects.

sa
Return array of long axes of objects according to shape_method

sao
Return array of long axes of objects according to shape_method

shape
Return list of shapely objects.

shape_method
Set or returns shape methods of all objects.

width
Returns width of extent.

class polylx.core.**PolyShape**(*shape*, *name*, *fid*)

Bases: object

Base class to store polygon or polyline

Properties: shape: shapely.geometry object name: name of polygon or polyline. fid: feature id

Note that all properties from shapely.geometry object are inherited.

__init__(*shape*, *name*, *fid*)

affine_transform(*matrix*)

Returns a transformed geometry using an affine transformation matrix. The matrix is provided as a list or tuple with 6 items: [a, b, d, e, xoff, yoff] which defines the equations for the transformed coordinates: x' = a * x + b * y + xoff y' = d * x + e * y + yoff

boundary_segments()

Create Boundaries from object boundary segments.

Example:

```
>>> g = Grains.from_shp()
>>> b = g.boundaries()
>>> bs1 = g[10].boundary_segments()
>>> bs2 = b[10].boundary_segments()
```

contains(*other*)

Returns True if the geometry contains the other, else False

crosses(*other*)

Returns True if the geometries cross, else False

difference(*other*)

Returns the difference of the geometries

disjoint(*other*)

Returns True if geometries are disjoint, else False

distance(*other*)

Unitless distance to other geometry (float)

dp(***kwargs*)

Douglas–Peucker simplification.

Keywords: tolerance: All points in the simplified object will be within the tolerance distance of the original geometry. Default Auto

equals(*other*)

Returns True if geometries are equal, else False

equals_exact(*other*, *tolerance*)

Returns True if geometries are equal to within a specified tolerance

feret(*angle=0*)

Returns the ferret diameter for given angle.

Args: angle: angle of caliper rotation

intersection(*other*)

Returns the intersection of the geometries

intersects(*other*)

Returns True if geometries intersect, else False

overlaps(*other*)

Returns True if geometries overlap, else False

paror(*angles=range(0, 180)*, *normalized=True*)

Returns paror function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

proj(*angle=0*)
Returns the cumulative projection of object for given angle.

Args: angle: angle of projection line

relate(*other*)
Returns the DE-9IM intersection matrix for the two geometries (string)

rotate(*angle*, ***kwargs*)
Returns a rotated geometry on a 2D plane. The angle of rotation can be specified in either degrees (default) or radians by setting use_radians=True. Positive angles are counter-clockwise and negative are clockwise rotations. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point.

scale(***kwargs*)
Returns a scaled geometry, scaled by factors along each dimension. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or coordinate tuple (x0, y0) for fixed point. Negative scale factors will mirror or reflect coordinates.

skew(***kwargs*)
Returns a skewed geometry, sheared by angles 'xs' along x and 'ys' along y direction. The shear angle can be specified in either degrees (default) or radians by setting use_radians=True. The point of origin can be a keyword 'center' for the object bounding box center (default), 'centroid' for the geometry's centroid, or a coordinate tuple (x0, y0) for fixed point.

surfor(*angles=range(0, 180)*, *normalized=True*)
Returns surfor function values. When normalized maximum value is 1 and correspond to max feret.

Args: angles: iterable angle values. Defaut range(180) normalized: whether to normalize values. Defaut True

symmetric_difference(*other*)
Returns the symmetric difference of the geometries (Shapely geometry)

touches(*other*)
Returns True if geometries touch, else False

translate(***kwargs*)
Returns a translated geometry shifted by offsets 'xoff' along x and 'yoff' along y direction.

union(*other*)
Returns the union of the geometries (Shapely geometry)

within(*other*)
Returns True if geometry is within the other, else False

ar
Returns axial ratio

Note that axial ratio is calculated from long and short axes calculated by actual `shape method`.

area
Area of the shape. For boundary returns 0.

bounds
Returns minimum bounding region (minx, miny, maxx, maxy)

centroid
Returns the geometric center of the object

length
Unitless length of the geometry (float)

ma
Returns mean axis

Mean axis is calculated as square root of long axis multiplied by short axis. Both axes are calculated by actual `shape method`.

representative_point
Returns a cheaply computed point that is guaranteed to be within the object.

shape_method
Returns shape method in use

class polylx.core.**Sample**(*name=''*)
Bases: object

Class to store both `Grains` and `Boundaries` objects

Properties: g: Grains object b: Boundaries.objects T. `networkx.Graph` storing grain topology

__init__(*name=''*)

bids(*idx*, *name=None*)

classmethod from_grains(*grains*, *name=''*)

classmethod from_shp(*filename='/home/docs/checkouts/readthedocs.org/user_builds/polylx/checkouts/develop/polyl*
phasefield='phase', *name=''*)

neighbors(*idx*, *name=None*, *inc=False*)
Returns array of indexes of neighbouring grains.

If name keyword is provided only neighbours with given name are returned.

neighbors_dist(*show=False*, *name=None*)
Return array of nearest neighbors distances.

If name keyword is provided only neighbours with given name are returned. When keyword show is True, plot is produced.

plot(***kwargs*)
Plot overlay of `Grains` and `Boundaries` of `Sample` object.

Args: alpha: Grains transparency. Default 0.8 pos: legend position "top" or "right". Defalt Auto ncol: number of columns for legend. show_fid: Show FID of objects. Default False show_index: Show index of objects. Default False

Returns matplotlib axes object.

show(***kwargs*)
Show plot of `Sample` objects.

triplets()

1.2 reports module

Generate simple pdf reports. Need rst2pdf tool (https://code.google.com/p/rst2pdf) to be installed.

Created on Wed Feb 5 21:42:54 2014

@author: Ondrej Lexa

Example: from polylx import * from polylx.reports import Report

g = Grains.from_shp()

fig, ax = plt.subplots() x = np.linspace(-8,8,200) ax.plot(x,np.sin(x))

r = Report('Test report') r.add_chapter('Things will start here') r.savefig(fig, width='75%') r.table([[1,2,120],[2,6,213],[3,4,118]],

title='Table example', header=['No','Val','Age'])

r.grainmap(g, width='75%') r.write_pdf()

class polylx.reports.**Report**(*title='Report'*)

Bases: object

__init__(*title='Report'*)

add_chapter(*title*)

add_section(*title*)

add_subsection(*title*)

dataframe(*df*, *title='Table'*, *header=None*, *format=None*, *stub_columns=None*, *widths=None*)

figure(*filename*, *width=None*, *height=None*)

fin()

matplotlib_fig(*fig*, *width=None*, *height=None*, *bbox_inches='tight'*, *dpi=150*)

pagebreak()

plot(*g*, *legend=None*, *loc='auto'*, *alpha=0.8*, *dpi=150*, *width=None*, *height=None*)

table(*rows*, *title='Table'*, *header=None*, *format=None*, *stub_columns=None*, *widths=None*)

transition()

write_pdf(*file='report.pdf'*)

write_rst(*file='report.rst'*)

CHAPTER 2

Tutorial

The microstructural analysis is a powerful, but underused tool of petrostructural analysis. Except acquirement of common statistical parameters, this technique can significantly improve understanding of processes of grain nucleation and grain growth, can bring insights on the role of surface energies or quantify duration of metamorphic and magmatic cooling events as long as appropriate thermodynamical data for studied mineral exist. This technique also allows systematic evaluation of degree of preferred orientations of grain boundaries in conjunction with their frequencies. This may help to better understand the mobility of grain boundaries and precipitations or removal of different mineral phases.

We introduce a new platform, object-oriented Python package PolyLX providing several core routines for data exchange, visualization and analysis of microstructural data, which can be run on any platform supported by Scientific Python environment.

2.1 Basic usage

To start working with **PolyLX** we need to import polylx package. For convinience, we will import polylx into actual namespace:

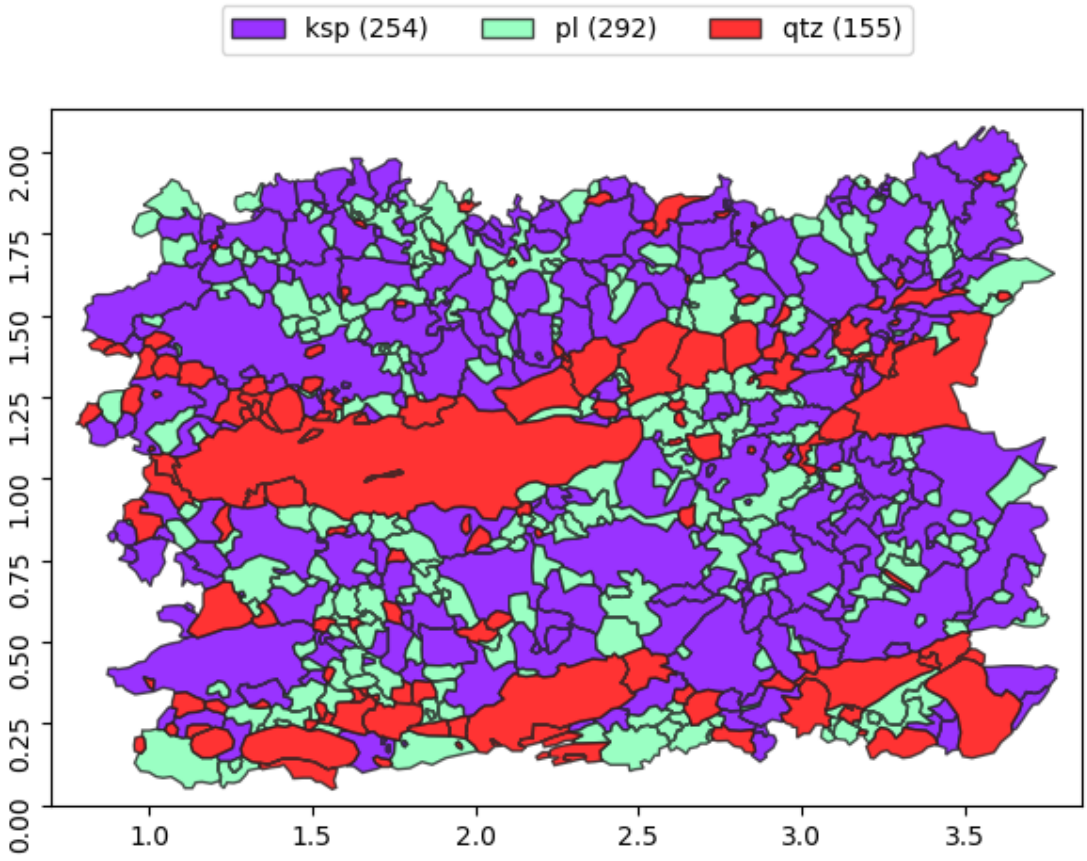
```
>>> from polylx import *
```

To read example data, we can use `Grains.from_shp` method without arguments. Note that we create new `Grains` object, which store all imported features (polygons) from shapefile:

```
>>> g = Grains.from_shp()
```

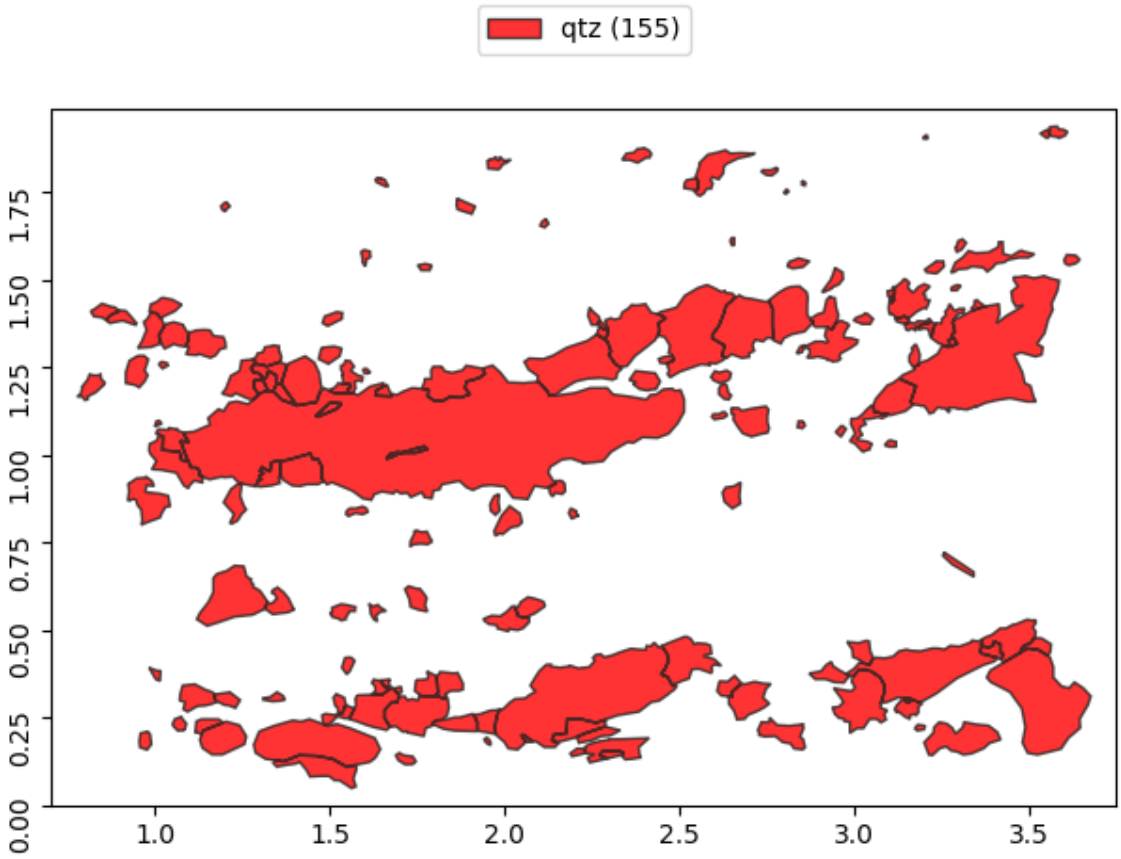
To visualize grain objects from shape file, we can use `show` method of Grains object:

```
>>> g.show()
```



To show only 'qtz' phase, we can use fancy indexing:

```
>>> g['qtz'].show()
```



Grains support dot notation to access individual properties. Note that most of properties are returned as `numpy.array`:

```
>>> g['qtz'].ar  # get axial ratios
array([ 1.46370088,  3.55371458,  1.43641139,  1.26293055,  2.10676277,
        1.45200805,  1.98973326,  1.97308557,  2.13420187,  1.76682269,
        1.70083897,  1.38205897,  1.88811465,  1.59948827,  2.50452919,
        1.60296389,  1.4918233 ,  2.15318719,  1.27665794,  1.38714959,
        1.67235338,  2.33179583,  1.30609967,  2.73148246,  1.02760669,
        1.33627299,  2.65451284,  1.29069569,  1.73051094,  1.25763409,
        1.90027316,  2.56110638,  1.78555385,  2.40926108,  2.26741705,
        1.71957235,  1.79168709,  1.04770164,  1.293186  ,  1.29420065,
        1.48331817,  2.15510614,  2.21246419,  1.57101091,  2.01989715,
        1.1428675 ,  2.02888455,  4.07405108,  1.47968881,  1.24770095,
        1.4750185 ,  1.37946472,  1.49048108,  1.56668345,  1.43717521,
        1.59756777,  1.58948843,  2.12557437,  2.54316052,  1.98917177,
        1.29809155,  1.70022052,  1.40121941,  1.24674038,  1.50255058,
        1.42880415,  1.73447054,  2.3548111 ,  1.52891827,  3.26773221,
        1.33011244,  2.26173396,  3.2151532 ,  2.15638456,  1.61602624,
        1.13898611,  2.91625233,  1.94275485,  2.68487563,  1.12446842,
        1.48814907,  1.79425743,  1.19512385,  1.28301942,  1.39853133,
        1.59860483,  3.80709622,  1.75016693,  1.59940152,  1.43972155,
        1.09439109,  2.00023212,  1.87470191,  1.04157011,  1.48561371,
        1.14172901,  1.48211332,  1.52569202,  1.59357336,  1.58054224,
        1.86890813,  1.84729576,  1.45085424,  1.4400654 ,  2.6284034 ,
        1.62077026,  1.35218688,  1.69040095,  1.2829313 ,  2.7380623 ,
        1.55901231,  1.72569674,  1.18396915,  1.67864861,  2.40971617,
        2.08496427,  2.12907657,  1.20981316,  1.46045276,  1.55428179,
        4.6980536 ,  2.32570855,  1.95106722,  1.81174297,  4.08295286,
        2.04530043,  1.56215221,  1.42587721,  1.70016792,  1.78887212,
        2.17273986,  2.47995119,  4.59660941,  3.43961286,  3.04193405,
        2.91162332,  2.98790473,  2.55352686,  1.33076709,  7.09385883,
```

```
        1.91715238,  1.47161362,  2.39020581,  1.51938795,  1.87839843,
        1.9946499 ,  2.27873759,  4.50321651,  5.78162231,  6.9806063 ,
        1.3177092 ,  2.33701528,  1.86371784,  1.26166336,  1.28322623])
```

More conviniet way to work with Grains attributes is collect any properties to `pandas.DataFrame` using `df` method:

```
>>> g.df('la', 'sa', 'lao', 'area', 'length', 'ead', 'ar').head(10)
           la        sa         lao      area    length       ead        ar
fid
0    0.066027  0.045110   70.596636  0.002286  0.186196  0.053956  1.463701
1    0.099033  0.057029   70.983857  0.004409  0.258753  0.074922  1.736522
2    0.074248  0.020893   61.438248  0.001123  0.175821  0.037813  3.553715
3    0.045232  0.031489   85.088587  0.001005  0.134427  0.035779  1.436411
4    0.136445  0.108038  170.839835  0.011489  0.398558  0.120948  1.262931
5    0.073578  0.044938  123.223347  0.002471  0.201258  0.056090  1.637319
6    0.103567  0.065119  149.397514  0.005213  0.283110  0.081474  1.590441
7    0.103189  0.077988   23.758847  0.005951  0.318774  0.087048  1.323142
8    0.187049  0.036611   82.108720  0.004407  0.404066  0.074904  5.109041
9    0.270513  0.128402   76.193288  0.024576  0.729051  0.176894  2.106763
```

Once you have `pandas.DataFrame`, check pandas manual to what you can do. Here is fe examples:

```
>>> g.df('ead').describe()
              ead
count  701.000000
mean     0.072812
std      0.056812
min      0.000350
25%      0.037140
50%      0.058338
75%      0.093503
max      0.638144
```

`agg` method aggregate properties according to defined classification (name by default):

```
>>> g.agg('area','sum', 'ead', 'mean', 'name', 'count')
                area       ead  name
name_class
ksp         2.443733  0.089710   254
pl          1.083516  0.060629   292
qtz         1.166097  0.068071   155
```

The `groups` method return `pandas.GroupBy` object which allows any pandas-style manipulation:

```
>>> g.groups('ead', 'area', 'la', 'sa').describe().T
name_class           ksp          pl           qtz
area count  2.540000e+02  292.000000  1.550000e+02
     mean   9.620995e-03    0.003711  7.523208e-03
     std    1.548182e-02    0.004170  2.778736e-02
     min    3.464873e-07    0.000003  9.629176e-08
     25%    1.341681e-03    0.001148  6.930225e-04
     50%    4.304819e-03    0.002289  1.805471e-03
     75%    1.115444e-02    0.004694  4.892680e-03
     max    1.323812e-01    0.028416  3.198359e-01
ead  count  2.540000e+02  292.000000  1.550000e+02
     mean   8.970974e-02    0.060629  6.807125e-02
     std    6.495077e-02    0.032438  7.054971e-02
     min    6.641998e-04    0.001850  3.501464e-04
     25%    4.133005e-02    0.038226  2.970151e-02
     50%    7.403298e-02    0.053984  4.794577e-02
     75%    1.191733e-01    0.077308  7.892656e-02
```

```
     max    4.105520e-01    0.190210  6.381439e-01
la   count  2.540000e+02  292.000000  1.550000e+02
     mean   1.295772e-01    0.086681  1.019395e-01
     std    1.053259e-01    0.053220  1.366152e-01
     min    1.013949e-03    0.006461  1.017291e-03
     25%    5.439610e-02    0.050202  4.314167e-02
     50%    9.871911e-02    0.072777  7.151284e-02
     75%    1.793952e-01    0.106761  1.206513e-01
     max    8.097226e-01    0.279398  1.437277e+00
sa   count  2.540000e+02  292.000000  1.550000e+02
     mean   7.545538e-02    0.049585  5.255111e-02
     std    5.428555e-02    0.027663  4.632415e-02
     min    3.648908e-04    0.000583  1.457310e-04
     25%    3.370683e-02    0.031980  2.183093e-02
     50%    6.643814e-02    0.043545  3.640605e-02
     75%    1.021515e-01    0.063468  6.490102e-02
     max    3.252086e-01    0.166726  3.035541e-01
```

The `classify` method could be used to define new classification, based on any property and using variety of methods:

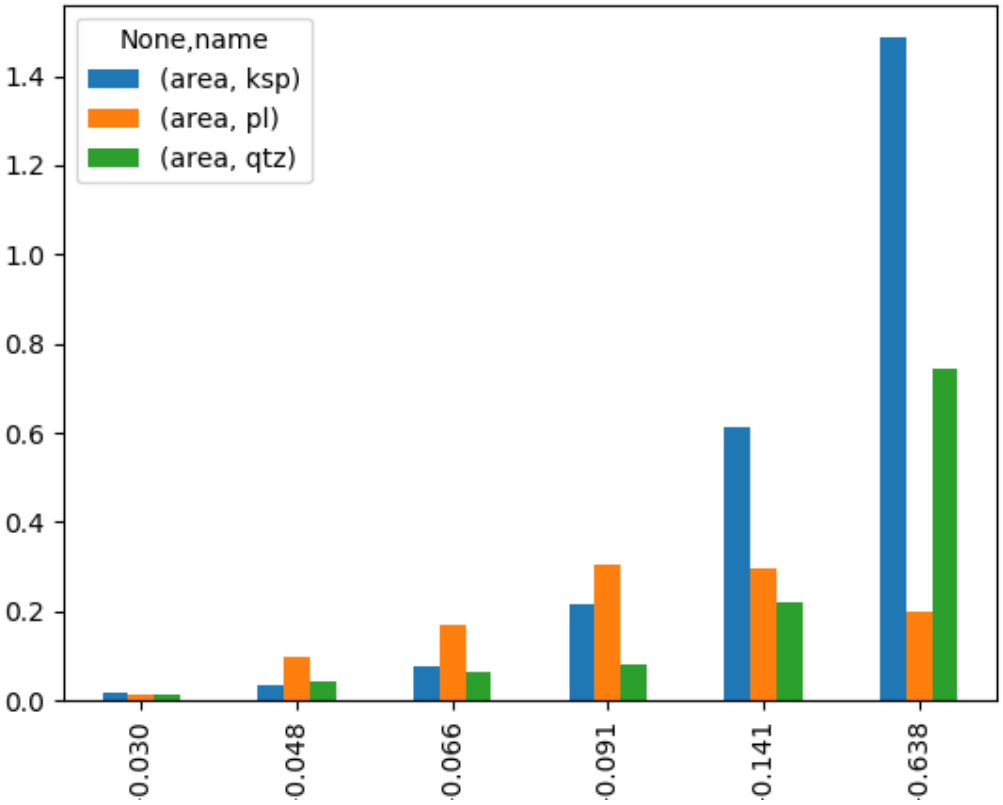
```
>>> g.classify('ead', k=6)
>>> df = g.df('class', 'name', 'area')
>>> df.head()
       ead_class name      area
fid
0    0.048-0.066  qtz  0.002286
1    0.066-0.091   pl  0.004409
2    0.030-0.048  qtz  0.001123
3    0.030-0.048  qtz  0.001005
4    0.091-0.141  qtz  0.011489
```

To summarize results for individual phases per class we can use `pandas.pivot_table`:

```
>>> pd.pivot_table(df,index=['ead_class'], columns=['name'], aggfunc=np.sum)
                area
name             ksp        pl       qtz
ead_class
0.000-0.030  0.017510  0.015057  0.015377
0.030-0.048  0.035587  0.096870  0.043866
0.048-0.066  0.077185  0.170371  0.065184
0.066-0.091  0.214921  0.305016  0.079672
0.091-0.141  0.612776  0.296543  0.218996
0.141-0.638  1.485754  0.199659  0.743003
```

or we can directly plot it:

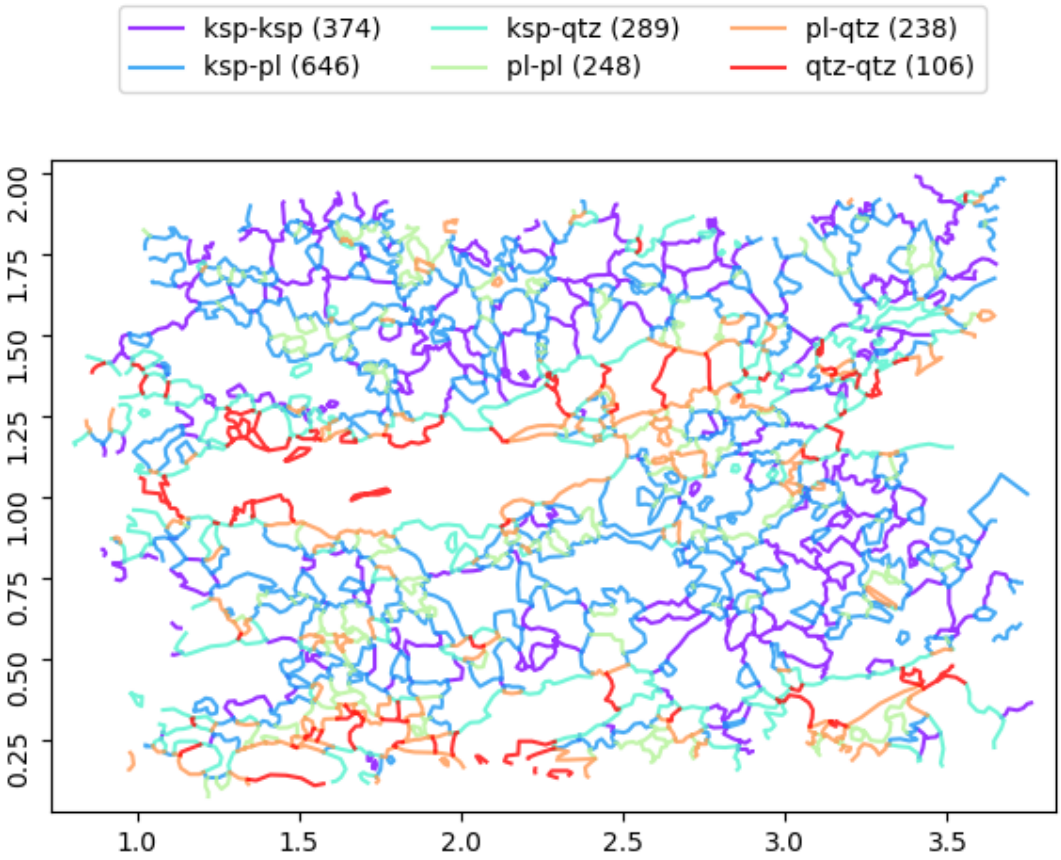
```
>>> pd.pivot_table(df,index=['ead_class'], columns=['name'], aggfunc=np.sum).
↪plot(kind='bar')
```



2.2 Work with boundaries

The `Boundaries` object could be created from grains with correct topology (use OpenJUMP, QGIS or ArcGIS to validate grain shapefile topology):

```
>>> b = g.boundaries()
>>> b.show()
```



Most of methods and properties demonstrated for `Grains` are valid also for boundaries:

```
>>> b.agg('sum', 'length')
               length
name_class
ksp-ksp     23.383974
ksp-pl      38.592227
ksp-qtz     17.920424
pl-pl       11.302490
pl-qtz      11.535006
qtz-qtz      6.617133
```

CHAPTER 3

Installation

At the command line:

```
$ easy_install polylx
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv polylx
$ pip install polylx
```

With conda you can install from personal channel:

```
$ conda install -c ondrolexa polylx
```

CHAPTER 4

Usage

To use PolyLX in a project:

```
import polylx
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://github.com/ondrolexa/polylx/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with “feature” is open to whoever wants to implement it.

5.1.4 Write Documentation

PolyLX could always use more documentation, whether as part of the official PolyLX docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/ondrolexa/polylx/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *polylx* for local development.

1. Fork the *polylx* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/polylx.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv polylx
$ cd polylx/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 polylx tests
$ python setup.py test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.

3. The pull request should work for Python 2.6, 2.7, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/ondrolexa/polylx/pull_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_polylx
```

CHAPTER 6

Credits

6.1 Development Lead

- Ondrej Lexa <lexa.ondrej@gmail.com>

6.2 Contributors

None yet. Why not be the first?

CHAPTER 7

Changes

7.1 0.1 (13 Feb 2015)

- First release

7.2 0.2 (18 Apr 2015)

- Smooth and simplify methods for Grains implemented
- Initial documentation added
- *phase* and *type* properties renamed to *name*

7.3 0.3 (22 Feb 2016)

7.3.1 0.3.1 (22 Feb 2016)

- classification is persitant trough fancy indexing
- empty classes allowed
- bootstrap method added to PolySet

7.3.2 0.3.2 (04 Jun 2016)

- PolyShape name forced to be string
- Creation of boundaries is Grains method

7.4 0.4 (20 Jun 2016)

- Sample neighbors_dist method to calculate neighbors distances

- Grains and Boundaries nndist to calculate nearest neighbors distances
- Fancy indexing with slices fixed
- Affine transformations affine_transform, rotate, scale, skew, translate methods implemented for Grains and Boundaries
- Sample name atribute added
- Sample bids method to get boundary id's related to grain added

7.4.1 0.4.1 (20 Jun 2016)

- Examples added to distribution

7.4.2 0.4.2 (02 Sep 2016)

- Sample has pairs property(dictionary) to map boundary id to grains id
- Sample triplets method returns list of grains id creating triple points

7.4.3 0.4.3 (02 Sep 2016)

- IPython added to requirements

7.4.4 0.4.4 (12 Jan 2017)

- Added MAEE (minimum area enclosing ellipse) to grain shape methods
- Removed embedded IPython and IPython requirements

7.4.5 0.4.5 (12 Jan 2017)

- shell script ipolylx opens interactive console

7.4.6 0.4.6 (04 Mar 2017)

- added plots module (initial)
- representative_point for Grains implemented
- moments calculation including holes
- surfor and parror functions added
- orientation of polygons is unified and checked
- minbox shape method added

7.4.7 0.4.8 (04 Mar 2017)

- bugfix

7.4.8 0.4.9 (12 Dec 2017)

- getindex method of Grains and Boundaries implemented
- Grain cdist property return centroid-vertex distance function
- Grain cdir property return centroid-vertex direction function
- Grain shape_vector property returns normalized Fourier descriptors
- Grain regularize method returns Grain with regularly distributed vertices
- Classification could be based on properties or any other values
- boundary_segments method added
- Smoothing, simplification and regularization of boundaries implemented
- Colortable for legend is persistant trough indexing. Classify method could be used to change it
- Default color table is seaborn muted for unique classification and matplotlib viridis for continuous classes

7.5 0.5 (XX YYY 2017)

- rose plot groupped according to classification
- get_class, class_iter methods added to Grains and Boundaries
- seaborn added to requirements
- several seaborn categorical plots are added as methods (swarmplot, boxplot, barplot, countplot)

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X