polarTransform Documentation

Release 2.0.0

Addison Elliott

Jan 09, 2019

Contents:

CHAPTER 1

Getting Started

1.1 Introduction

polarTransform is a Python package for converting images between the polar and Cartesian domain. It contains many features such as specifying the start/stop radius and angle, interpolation order (bicubic, linear, nearest, etc), and much more.

1.2 License

polarTransform has an MIT-based license.

1.3 Installing

1.3.1 Prerequisites

- Python 3
- **Dependencies:**
  - numpy
  - scipy
  - scikit-image

1.3.2 Installing polarTransform

polarTransform is currently available on PyPi. The simplest way to install alone is using `pip` at a command line:

```
pip install polarTransform
```

which installs the latest release. To install the latest code from the repository (usually stable, but may have undocumented changes or bugs):

```
pip install git+https://github.com/addisonElliott/polarTransform.git
```

For developers, you can clone the pydicom repository and run the `setup.py` file. Use the following commands to get a copy from GitHub and install all dependencies:

```
git clone pip install git+https://github.com/addisonElliott/polarTransform.git
cd polarTransform
pip install .
```

or, for the last line, instead use:

```
pip install -e .
```

to install in 'develop' or 'editable' mode, where changes can be made to the local working code and Python will use the updated polarTransform code.

1.4 Test and coverage

Run the following command in the base directory to run the tests:

```
python -m unittest discover -v polarTransform/tests
```

1.5 Using polarTransform

Once installed, the package can be imported at a Python command line or used in your own Python program with `import polarTransform`. See the *User Guide* for more details of how to use the package.

1.6 Support

Bugs can be submitted through the issue tracker.

Pull requests are welcome too!

1.7 Next Steps

To start learning how to use polarTransform, see the *User Guide*.

CHAPTER 2

User Guide

`convertToPolarImage` and `convertToCartesianImage` are the two primary functions that make up this package. The two functions are opposites of one another, reversing the action that the other function does.

As the names suggest, the two functions convert an image from the cartesian or polar domain to the other domain with a given set of parameters. The power of these functions is that the user can specify the resulting image resolution, interpolation order, initial and final radii or angles and much much more. See the *Reference Guide* for more information on the specific parameters that are supported.

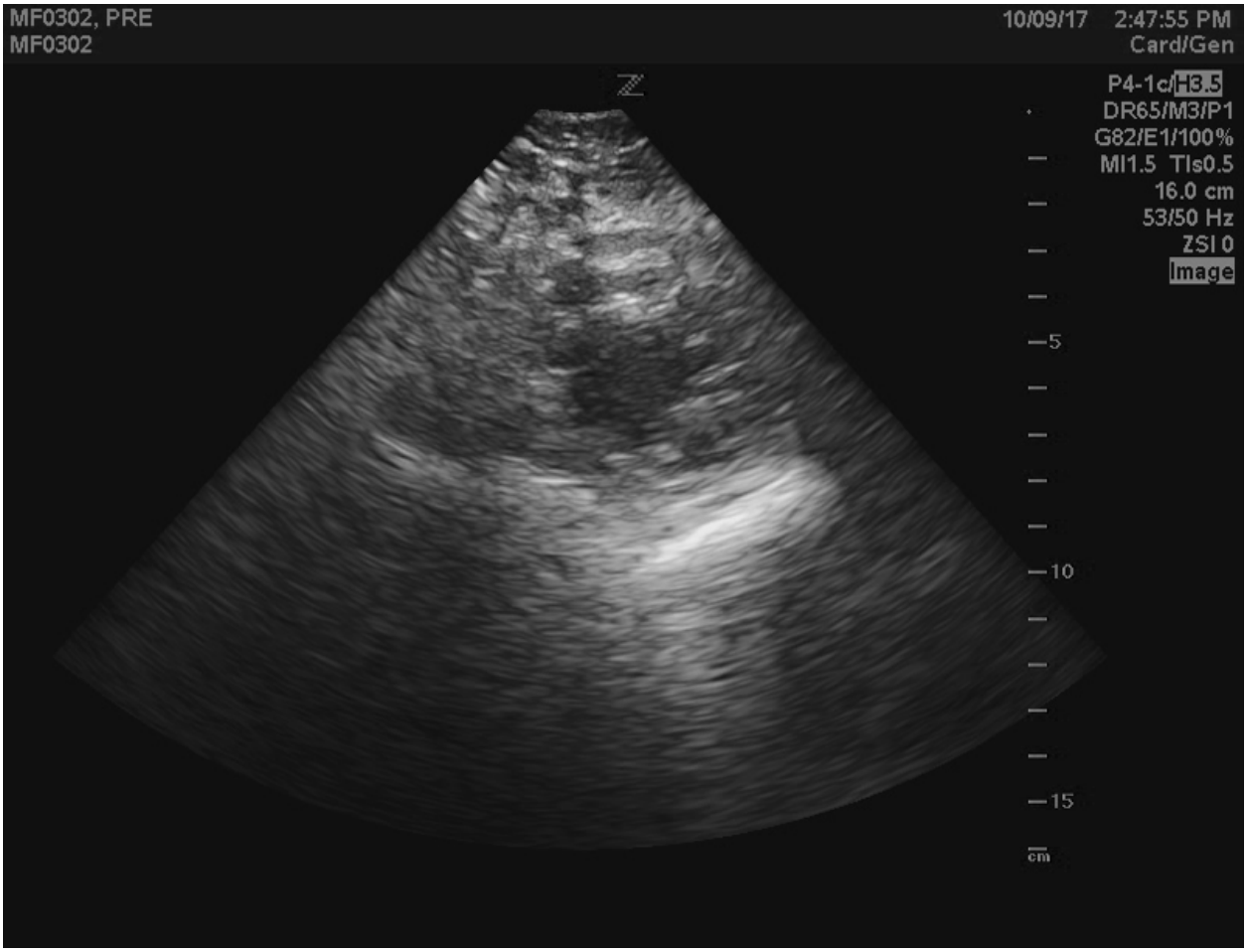
Since there are quite a few parameters that can be specified for the conversion functions, the class `ImageTransform` is created and returned from the `convertToPolarImage` or `convertToCartesianImage` functions (along with the converted image) that contains the arguments specified. The benefit of this class is that if one wants to convert the image back to another domain or convert points on either image to/from the other domain, they can simply call the functions within the `ImageTransform` class without specifying all of the arguments again.

The examples below use images from the test suite. The code snippets should run without modification except for changing the paths to point to the correct image.

2.1 Example 1

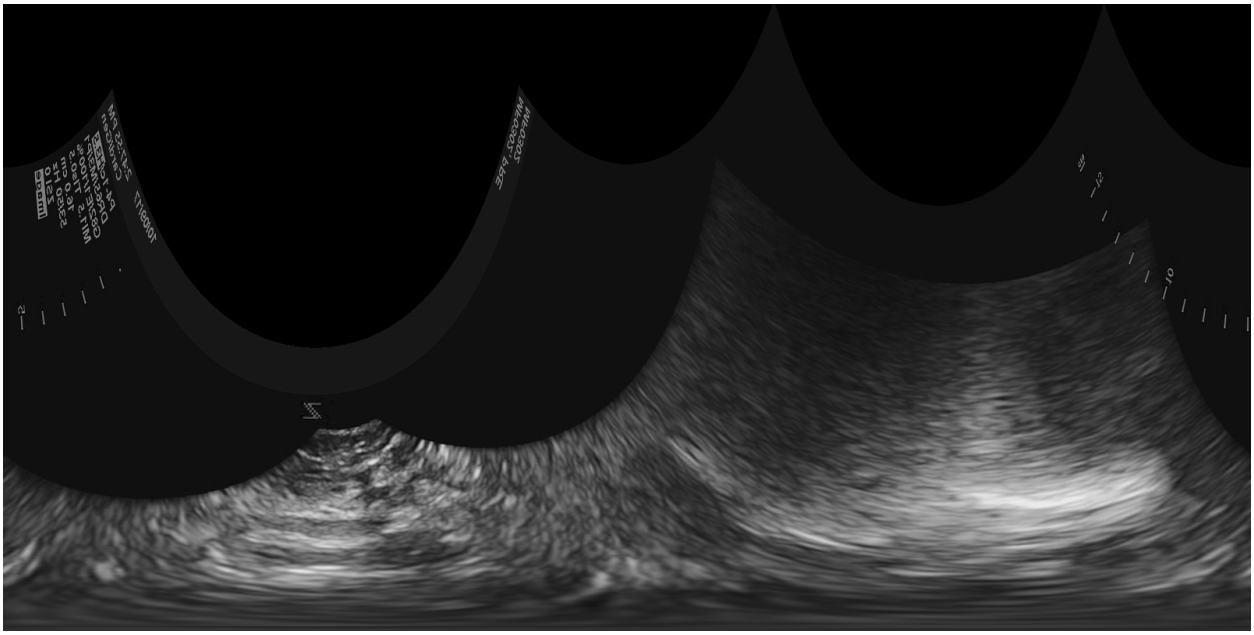
Let us take a B-mode echocardiogram and convert it to the polar domain. This is essentially reversing the scan conversion done internally by the ultrasound machine.

Here is the B-mode image:




```
import polarTransform
import matplotlib.pyplot as plt
import imageio

cartesianImage = imageio.imread('IMAGE_PATH_HERE')

polarImage, ptSettings = polarTransform.convertToPolarImage(cartesianImage,␣
↪center=[401, 365])
plt.imshow(polarImage.T, origin='lower')
```

Resulting polar domain image:



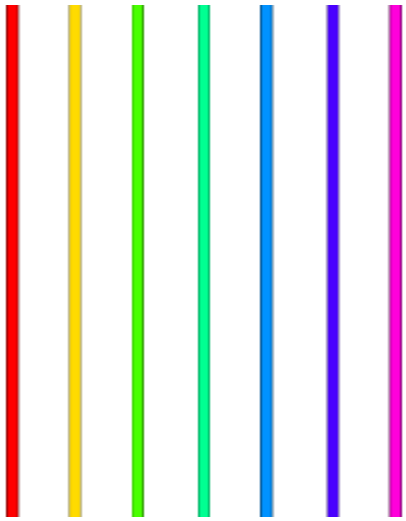
The example input image has a width of 800px and a height of 604px. Since many imaging libraries use C-order rather than Fortran order, the Numpy array containing the image data loaded from imageio has a shape of (604, 800). This is what polarTransform expects for an image where the first dimension is the slowest varying (y) and the last dimension is the fastest varying (x). Additional dimensions can be present before the y & x dimensions in which case polarTransform will transform each 2D slice individually.

The center argument should be a list, tuple or Numpy array of length 2 with format (x, y). A common theme throughout this library is that points will be specified in Fortran order, i.e. (x, y) or (r, theta) whilst data and image sizes will be specified in C-order, i.e. (y, x) or (theta, r).

The polar image returned in this example is in C-order. So this means that the data is (theta, r). When displaying an image using matplotlib, the first dimension is y and second is x. The image is transposed before displaying to flip it 90 degrees.

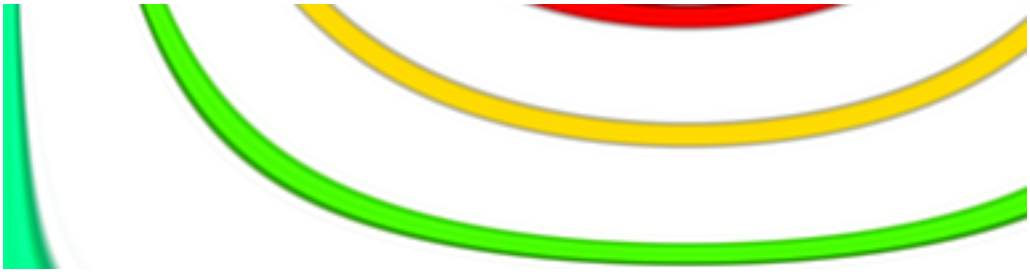
2.2 Example 2

Input image:



```
import polarTransform
import matplotlib.pyplot as plt
import imageio

verticalLinesImage = imageio.imread('IMAGE_PATH_HERE')

polarImage, ptSettings = polarTransform.convertToPolarImage(verticalLinesImage,␣
↪initialRadius=30,
                                                            finalRadius=100,␣
↪initialAngle=2 / 4 * np.pi,
                                                            finalAngle=5 / 4 * np.pi,␣
↪hasColor=True)

cartesianImage = ptSettings.convertToCartesianImage(polarImage)

plt.figure()
plt.imshow(polarImage.T, origin='lower')

plt.figure()
plt.imshow(cartesianImage, origin='lower')
```

Resulting polar domain image:



Converting back to the cartesian image results in:

Once again, when displaying polar images using matplotlib, the image is first transposed to rotate the image 90 degrees. This makes it easier to view the image because the theta dimension is longer than the radial dimension.

The hasColor argument was set to True in this example because the image contains color images. The example RGB image has a width and height of 256px. The shape of the image loaded via imageio package is (256, 256, 3). By specified hasColor=True, the last dimension will be shifted to the front and then the polar transformation will occur on each channel separately. Before returning, the function will shift the channel dimension back to the end. If a RGBA image is loaded, it is advised to only transform the first 3 channels and then set the alpha channel to fully on.

CHAPTER 3

Reference Guide

3.1 Table of Contents

3.2 polarTransform Module

polarTransform.**convertToCartesianImage**(*image*, *center=None*, *initialRadius=None*, *finalRadius=None*, *initialAngle=None*, *finalAngle=None*, *imageSize=None*, *hasColor=False*, *order=3*, *border='constant'*, *borderVal=0.0*, *useMultiThreading=False*, *settings=None*)

Convert polar image to cartesian image.

Using a polar image, this function creates a cartesian image. This function is versatile because it can automatically calculate an appropriate cartesian image size and center given the polar image. In addition, parameters for converting to the polar domain are necessary for the conversion back to the cartesian domain.

Parameters

image [N-dimensional numpy.ndarray] Polar image to convert to cartesian domain

Image should be structured in C-order, i.e. the axes should be ordered (..., z, theta, r, [ch]). The channel axes should only be present if hasColor is True. This format is arbitrary

but is selected to stay consistent with the traditional C-order representation in the Cartesian domain.

In the mathematical domain, Cartesian coordinates are traditionally represented as (x, y, z) and as (r, theta, z) in the polar domain. When storing Cartesian data in C-order, the axes are usually flipped and the data is saved as (z, y, x). Thus, the polar domain coordinates are also flipped to stay consistent, hence the format (z, theta, r).

Note: For multi-dimensional images above 2D, the cartesian transformation is applied individually across each 2D slice. The last two dimensions should be the r & theta dimensions, unless `hasColor` is True in which case the 2nd and 3rd to last dimensions should be. The multidimensional shape will be preserved for the resulting cartesian image (besides the polar dimensions).

---

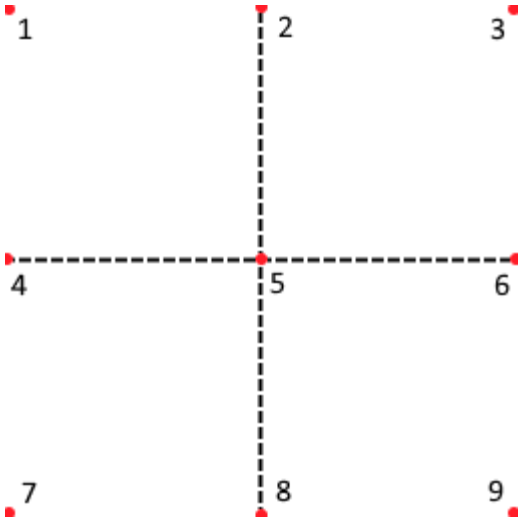
center [`str` or (2,) `list`, `tuple` or `numpy.ndarray` of `int`, optional] Specifies the center in the cartesian image to use as the origin in polar domain. The center in the cartesian domain will be (0, 0) in the polar domain.

If center is not set, then it will default to `middle-middle`. If the image size is `None`, the center is calculated after the image size is determined.

For relative positioning within the image, center can be one of the string values in the table below. The quadrant column contains the visible quadrants for the given center. initialAngle and finalAngle must contain at least one of the quadrants, otherwise an error will be thrown because the resulting cartesian image is blank. An example cartesian image is given below with annotations to what the center will be given a center string.

Table 2: Valid center strings

Value	Quadrant	Location in image
top-left	IV	1
top-middle	III, IV	2
top-right	III	3
middle-left	I, IV	4
middle-middle	I, II, III, IV	5
middle-right	II, III	6
bottom-left	I	7
bottom-middle	I, II	8
bottom-right	II	9



initialRadius [`int`, optional] Starting radius in pixels from the center of the cartesian image in the polar image

The polar image begins at this radius, i.e. the first row of the polar image corresponds to this starting radius.

If initialRadius is not set, then it will default to `0`.

finalRadius [`int`, optional] Final radius in pixels from the center of the cartesian image in the polar image

The polar image ends at this radius, i.e. the last row of the polar image corresponds to this ending radius.

Note: The polar image does **not** include this radius. It includes all radii starting from initial to final radii **excluding** the final radius. Rather, it will stop one step size before the final radius. Assuming the radial resolution (see `radiusSize`) is small enough, this should not matter.

If finalRadius is not set, then it will default to the maximum radius which is the size of the radial (1st) dimension of the polar image.

initialAngle [`float`, optional] Starting angle in radians in the polar image

The polar image begins at this angle, i.e. the first column of the polar image corresponds to this starting angle.

Radian angle is with respect to the x-axis and rotates counter-clockwise. The angle should be in the range of 0 to 2π.

If initialAngle is not set, then it will default to `0.0`.

finalAngle [`float`, optional] Final angle in radians in the polar image

The polar image ends at this angle, i.e. the last column of the polar image corresponds to this ending angle.

Note: The polar image does **not** include this angle. It includes all angles starting from initial to final angle **excluding** the final angle. Rather, it stops one step size before the final

angle. Assuming the angular resolution (see `angleSize`) is small enough, this should not matter.

Radian angle is with respect to the x-axis and rotates counter-clockwise. The angle should be in the range of 0 to 2π.

If finalAngle is not set, then it will default to 2π.

imageSize [(2,) `list`, `tuple` or `numpy.ndarray` of `int`, optional] Desired size of cartesian image where 1st dimension is number of rows and 2nd dimension is number of columns

If imageSize is not set, then it defaults to the size required to fit the entire polar image on a cartesian image.

hasColor [`bool`, optional] Whether or not the polar image contains color channels

This means that the image is structured as (..., y, x, ch) or (..., theta, r, ch) for Cartesian or polar images, respectively. If color channels are present, the last dimension (channel axes) will be shifted to the front, converted and then shifted back to its original location.

Default is `False`

Note: If an alpha band (4th channel of image is present), then it will be converted. Typically, this is unwanted, so the recommended solution is to transform the first 3 channels and set the 4th channel to fully on.

---

order [`int` (0-5), optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

The following orders have special names:

- 0 - nearest neighbor
- 1 - bilinear
- 3 - bicubic

border [{'constant', 'nearest', 'wrap', 'reflect'}, optional] Polar points outside the cartesian image boundaries are filled according to the given mode.

Default is 'constant'

The following table describes the mode and expected output when seeking past the boundaries. The input column is the 1D input array whilst the extended columns on either side of the input array correspond to the expected values for the given mode if one extends past the boundaries.

Table 3: Valid border modes and expected output

Mode	Ext.	Input	Ext.
mirror	4 3 2	1 2 3 4 5 6 7 8	7 6 5
reflect	3 2 1	1 2 3 4 5 6 7 8	8 7 6
nearest	1 1 1	1 2 3 4 5 6 7 8	8 8 8
constant	0 0 0	1 2 3 4 5 6 7 8	0 0 0
wrap	6 7 8	1 2 3 4 5 6 7 8	1 2 3

Refer to `scipy.ndimage.map_coordinates()` for more details on this argument.

borderVal [same datatype as `image`, optional] Value used for polar points outside the cartesian image boundaries if `border` = 'constant'.

Default is 0.0

useMultiThreading [`bool`, optional] Whether to use multithreading when applying transformation for 3D images. This considerably speeds up the execution time for large images but adds overhead for smaller 3D images.

Default is `False`

settings [*ImageTransform*, optional] Contains metadata for conversion between polar and cartesian image.

Settings contains many of the arguments in *convertToPolarImage()* and *convertToCartesianImage()* and provides an easy way of passing these parameters along without having to specify them all again.

> **Warning:** Cleaner and more succint to use *ImageTransform.convertToCartesianImage()*

If settings is not specified, then the other arguments are used in this function and the defaults will be calculated if necessary. If settings is given, then the values from settings will be used.

Returns

cartesianImage [N-dimensional `numpy.ndarray`] Cartesian image

Resulting image is structured in C-order, i.e. the axes are ordered as (..., z, y, x, [ch]). This format is the traditional method of storing image data in Python.

Resulting image shape will be the same as the input image except for the polar dimensions are replaced with the Cartesian dimensions.

settings [*ImageTransform*] Contains metadata for conversion between polar and cartesian image.

Settings contains many of the arguments in *convertToPolarImage()* and *convertToCartesianImage()* and provides an easy way of passing these parameters along without having to specify them all again.

polarTransform.**convertToPolarImage**(*image*, *center=None*, *initialRadius=None*, *finalRadius=None*, *initialAngle=None*, *finalAngle=None*, *radiusSize=None*, *angleSize=None*, *hasColor=False*, *order=3*, *border='constant'*, *borderVal=0.0*, *useMultiThreading=False*, *settings=None*)

Convert cartesian image to polar image.

Using a cartesian image, this function creates a polar domain image where the first dimension is radius and second dimension is the angle. This function is versatile because it allows different starting and stopping radii and angles to extract the polar region you are interested in.

Note: Traditionally images are loaded such that the origin is in the upper-left hand corner. In these cases the `initialAngle` and `finalAngle` will rotate clockwise from the x-axis. For simplicitly, it is recommended to flip the image along first dimension before passing to this function.

Parameters

image [N-dimensional `numpy.ndarray`] Cartesian image to convert to polar domain

Image should be structured in C-order, i.e. the axes should be ordered as (..., z, y, x, [ch]). This format is the traditional method of storing image data in Python.

Note: For multi-dimensional images above 2D, the polar transformation is applied individually across each 2D slice. The last two dimensions should be the x & y dimensions, unless `hasColor` is True in which case the 2nd and 3rd to last dimensions should be. The multidimensional shape will be preserved for the resulting polar image (besides the Cartesian dimensions).

---

center [(2,) `list`, `tuple` or `numpy.ndarray` of `int`, optional] Specifies the center in the cartesian image to use as the origin in polar domain. The center in the cartesian domain will be (0, 0) in the polar domain.

The center is structured as (x, y) where the first item is the x-coordinate and second item is the y-coordinate.

If center is not set, then it will default to `round(image.shape[::-1] / 2)`.

initialRadius [`int`, optional] Starting radius in pixels from the center of the cartesian image that will appear in the polar image

The polar image will begin at this radius, i.e. the first row of the polar image will correspond to this starting radius.

If initialRadius is not set, then it will default to `0`.

finalRadius [`int`, optional] Final radius in pixels from the center of the cartesian image that will appear in the polar image

The polar image will end at this radius, i.e. the last row of the polar image will correspond to this ending radius.

Note: The polar image will **not** include this radius. It will include all radii starting from initial to final radii **excluding** the final radius. Rather, it will stop one step size before the final radius. Assuming the radial resolution (see `radiusSize`) is small enough, this should not matter.

If finalRadius is not set, then it will default to the maximum radius of the cartesian image. Using the furthest corner from the center, the finalRadius can be calculated as:

$$finalRadius = \sqrt{((X_{max} - X_{center})^2 + (Y_{max} - Y_{center})^2)}$$

initialAngle [`float`, optional] Starting angle in radians that will appear in the polar image

The polar image will begin at this angle, i.e. the first column of the polar image will correspond to this starting angle.

Radian angle is with respect to the x-axis and rotates counter-clockwise. The angle should be in the range of 0 to 2π.

If initialAngle is not set, then it will default to `0.0`.

finalAngle [`float`, optional] Final angle in radians that will appear in the polar image

The polar image will end at this angle, i.e. the last column of the polar image will correspond to this ending angle.

Note: The polar image will **not** include this angle. It will include all angle starting from initial to final angle **excluding** the final angle. Rather, it will stop one step size before the final angle. Assuming the angular resolution (see `angleSize`) is small enough, this should not matter.

Radian angle is with respect to the x-axis and rotates counter-clockwise. The angle should be in the range of 0 to 2π.

If finalAngle is not set, then it will default to 2π.

radiusSize [`int`, optional] Size of polar image for radial (1st) dimension

This in effect determines the resolution of the radial dimension of the polar image based on the `initialRadius` and `finalRadius`. Resolution can be calculated using equation below in radial px per cartesian px:

$$radialResolution = \frac{radiusSize}{finalRadius - initialRadius}$$

If radiusSize is not set, then it will default to the minimum size necessary to ensure that image information is not lost in the transformation. The minimum resolution necessary can be found by finding the smallest change in radius from two connected pixels in the cartesian image. Through experimentation, there is a surprisingly close relationship between the maximum difference from width or height of the cartesian image to the `center` times two.

The radiusSize is calculated based on this relationship and is proportional to the `initialRadius` and `finalRadius` given.

angleSize [`int`, optional] Size of polar image for angular (2nd) dimension

This in effect determines the resolution of the angular dimension of the polar image based on the `initialAngle` and `finalAngle`. Resolution can be calculated using equation below in angular px per cartesian px:

$$angularResolution = \frac{angleSize}{finalAngle - initialAngle}$$

If angleSize is not set, then it will default to the minimum size necessary to ensure that image information is not lost in the transformation. The minimum resolution necessary can be found by finding the smallest change in angle from two connected pixels in the cartesian image.

For a cartesian image with either dimension greater than 500px, the angleSize is set to be **two** times larger than the largest dimension proportional to `initialAngle` and `finalAngle`. Otherwise, for a cartesian image with both dimensions less than 500px, the angleSize is set to be **four** times larger the largest dimension proportional to `initialAngle` and `finalAngle`.

Note: The above logic **estimates** the necessary angleSize to reduce image information loss. No algorithm currently exists for determining the required angleSize.

---

hasColor [`bool`, optional] Whether or not the cartesian image contains color channels

This means that the image is structured as (..., y, x, ch) or (..., theta, r, ch) for Cartesian or polar images, respectively. If color channels are present, the last dimension (channel axes) will be shifted to the front, converted and then shifted back to its original location.

Default is `False`

Note: If an alpha band (4th channel of image is present), then it will be converted. Typically, this is unwanted, so the recommended solution is to transform the first 3 channels and set the 4th channel to fully on.

---

order [`int` (0-5), optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

The following orders have special names:

- 0 - nearest neighbor
- 1 - bilinear
- 3 - bicubic

border [{'constant', 'nearest', 'wrap', 'reflect'}, optional] Polar points outside the cartesian image boundaries are filled according to the given mode.

Default is 'constant'

The following table describes the mode and expected output when seeking past the boundaries. The input column is the 1D input array whilst the extended columns on either side of the input array correspond to the expected values for the given mode if one extends past the boundaries.

Table 4: Valid border modes and expected output

Mode	Ext.	Input	Ext.
mirror	4 3 2	1 2 3 4 5 6 7 8	7 6 5
reflect	3 2 1	1 2 3 4 5 6 7 8	8 7 6
nearest	1 1 1	1 2 3 4 5 6 7 8	8 8 8
constant	0 0 0	1 2 3 4 5 6 7 8	0 0 0
wrap	6 7 8	1 2 3 4 5 6 7 8	1 2 3

Refer to `scipy.ndimage.map_coordinates()` for more details on this argument.

borderVal [same datatype as `image`, optional] Value used for polar points outside the cartesian image boundaries if `border` = 'constant'.

Default is 0.0

useMultiThreading [`bool`, optional] Whether to use multithreading when applying transformation for 3D images. This considerably speeds up the execution time for large images but adds overhead for smaller 3D images.

Default is `False`

settings [*`ImageTransform`*, optional] Contains metadata for conversion between polar and cartesian image.

Settings contains many of the arguments in *`convertToPolarImage()`* and *`convertToCartesianImage()`* and provides an easy way of passing these parameters along without having to specify them all again.

Warning: Cleaner and more succint to use *`ImageTransform.convertToPolarImage()`*

If settings is not specified, then the other arguments are used in this function and the defaults will be calculated if necessary. If settings is given, then the values from settings will be used.

Returns

polarImage [N-dimensional `numpy.ndarray`] Polar image

Resulting image is structured in C-order, i.e. the axes are be ordered as (..., z, theta, r, [ch]) depending on if the input image was 3D. This format is arbitrary but is selected to stay consistent with the traditional C-order representation in the Cartesian domain.

In the mathematical domain, Cartesian coordinates are traditionally represented as (x, y, z) and as (r, theta, z) in the polar domain. When storing Cartesian data in C-order, the axes are usually flipped and the data is saved as (z, y, x). Thus, the polar domain coordinates are also flipped to stay consistent, hence the format (z, theta, r).

Resulting image shape will be the same as the input image except for the Cartesian dimensions are replaced with the polar dimensions.

settings [*ImageTransform*] Contains metadata for conversion between polar and cartesian image.

Settings contains many of the arguments in *convertToPolarImage()* and *convertToCartesianImage()* and provides an easy way of passing these parameters along without having to specify them all again.

class polarTransform.**ImageTransform**(*center*, *initialRadius*, *finalRadius*, *initialAngle*, *finalAngle*, *cartesianImageSize*, *polarImageSize*, *hasColor*)

Class to store settings when converting between cartesian and polar domain

convertToCartesianImage(*image*, *order=3*, *border='constant'*, *borderVal=0.0*, *useMultiThreading=False*)

Convert polar image to cartesian image.

Using a polar image, this function creates a cartesian image. This function is versatile because it can automatically calculate an appropiate cartesian image size and center given the polar image. In addition, parameters for converting to the polar domain are necessary for the conversion back to the cartesian domain.

Parameters

image [N-dimensional `numpy.ndarray`] Polar image to convert to cartesian domain

Image should be structured in C-order, i.e. the axes should be ordered (..., z, theta, r, [ch]). The channel axes should only be present if `hasColor` is `True`. This format is arbitrary but is selected to stay consistent with the traditional C-order representation in the Cartesian domain.

In the mathematical domain, Cartesian coordinates are traditionally represented as (x, y, z) and as (r, theta, z) in the polar domain. When storing Cartesian data in C-order, the axes are usually flipped and the data is saved as (z, y, x). Thus, the polar domain coordinates are also flipped to stay consistent, hence the format (z, theta, r).

Note: For multi-dimensional images above 2D, the cartesian transformation is applied individually across each 2D slice. The last two dimensions should be the r & theta dimensions, unless `hasColor` is True in which case the 2nd and 3rd to last dimensions should be. The multidimensional shape will be preserved for the resulting cartesian image (besides the polar dimensions).

---

order [int (0-5), optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

The following orders have special names:

- 0 - nearest neighbor
- 1 - bilinear
- 3 - bicubic

border [{'constant', 'nearest', 'wrap', 'reflect'}, optional] Polar points outside the cartesian image boundaries are filled according to the given mode.

Default is 'constant'

The following table describes the mode and expected output when seeking past the boundaries. The input column is the 1D input array whilst the extended columns on either side of the input array correspond to the expected values for the given mode if one extends past the boundaries.

Table 5: Valid border modes and expected output

Mode	Ext.	Input	Ext.
mirror	4 3 2	1 2 3 4 5 6 7 8	7 6 5
reflect	3 2 1	1 2 3 4 5 6 7 8	8 7 6
nearest	1 1 1	1 2 3 4 5 6 7 8	8 8 8
constant	0 0 0	1 2 3 4 5 6 7 8	0 0 0
wrap	6 7 8	1 2 3 4 5 6 7 8	1 2 3

Refer to scipy.ndimage.map_coordinates() for more details on this argument.

borderVal [same datatype as image, optional] Value used for polar points outside the cartesian image boundaries if border = 'constant'.

Default is 0.0

useMultiThreading [bool, optional] Whether to use multithreading when applying transformation for 3D images. This considerably speeds up the execution time for large images but adds overhead for smaller 3D images.

Default is False

Returns

cartesianImage [N-dimensional numpy.ndarray] Cartesian image

Resulting image is structured in C-order, i.e. the axes are ordered as (..., z, y, x, [ch]). This format is the traditional method of storing image data in Python.

Resulting image shape will be the same as the input image except for the polar dimensions are replaced with the Cartesian dimensions.

See also:

convertToCartesianImage()

convertToPolarImage(*image*, *order=3*, *border='constant'*, *borderVal=0.0*, *useMultiThreading=False*)

Convert cartesian image to polar image.

Using a cartesian image, this function creates a polar domain image where the first dimension is radius and second dimension is the angle. This function is versatile because it allows different starting and stopping radii and angles to extract the polar region you are interested in.

Note: Traditionally images are loaded such that the origin is in the upper-left hand corner. In these cases the `initialAngle` and `finalAngle` will rotate clockwise from the x-axis. For simplicitly, it is recommended to flip the image along first dimension before passing to this function.

Parameters

image [N-dimensional `numpy.ndarray`] Cartesian image to convert to polar domain

Image should be structured in C-order, i.e. the axes should be ordered as (..., z, y, x, [ch]). This format is the traditional method of storing image data in Python.

Note: For multi-dimensional images above 2D, the polar transformation is applied individually across each 2D slice. The last two dimensions should be the x & y dimensions, unless `hasColor` is True in which case the 2nd and 3rd to last dimensions should be. The multidimensional shape will be preserved for the resulting polar image (besides the Cartesian dimensions).

order [`int` (0-5), optional] The order of the spline interpolation, default is 3. The order has to be in the range 0-5.

The following orders have special names:

- 0 - nearest neighbor
- 1 - bilinear
- 3 - bicubic

border [{'constant', 'nearest', 'wrap', 'reflect'}, optional] Polar points outside the cartesian image boundaries are filled according to the given mode.

Default is 'constant'

The following table describes the mode and expected output when seeking past the boundaries. The input column is the 1D input array whilst the extended columns on either side of the input array correspond to the expected values for the given mode if one extends past the boundaries.

Table 6: Valid border modes and expected output

Mode	Ext.	Input	Ext.
mirror	4 3 2	1 2 3 4 5 6 7 8	7 6 5
reflect	3 2 1	1 2 3 4 5 6 7 8	8 7 6
nearest	1 1 1	1 2 3 4 5 6 7 8	8 8 8
constant	0 0 0	1 2 3 4 5 6 7 8	0 0 0
wrap	6 7 8	1 2 3 4 5 6 7 8	1 2 3

Refer to `scipy.ndimage.map_coordinates()` for more details on this argument.

borderVal [same datatype as `image`, optional] Value used for polar points outside the cartesian image boundaries if `border` = 'constant'.

Default is 0.0

Returns

polarImage [N-dimensional numpy.ndarray] Polar image

Resulting image is structured in C-order, i.e. the axes are be ordered as (..., z, theta, r, [ch]) depending on if the input image was 3D. This format is arbitrary but is selected to stay consistent with the traditional C-order representation in the Cartesian domain.

In the mathematical domain, Cartesian coordinates are traditionally represented as (x, y, z) and as (r, theta, z) in the polar domain. When storing Cartesian data in C-order, the axes are usually flipped and the data is saved as (z, y, x). Thus, the polar domain coordinates are also flipped to stay consistent, hence the format (z, theta, r).

Resulting image shape will be the same as the input image except for the Cartesian dimensions are replaced with the polar dimensions.

getCartesianPointsImage(*points*)

Convert list of polar points from image to cartesian image points based on transform metadata

---

Note: This does **not** convert from polar to cartesian points, but rather converts pixels from polar image to pixels from cartesian image using *ImageTransform*.

The returned points are not rounded to the nearest point. User must do that by hand if desired.

Parameters

points [(N, 2) or (2,) numpy.ndarray] List of polar points to convert to cartesian domain

First column is r and second column is theta

Returns

cartesianPoints [(N, 2) or (2,) numpy.ndarray] Corresponding cartesian points from polar points using *ImageTransform*

See also:

getCartesianPointsImage(), getCartesianPoints(), getCartesianPoints2()

getPolarPointsImage(*points*)

Convert list of cartesian points from image to polar image points based on transform metadata

---

Note: This does **not** convert from cartesian to polar points, but rather converts pixels from cartesian image to pixels from polar image using *ImageTransform*.

The returned points are not rounded to the nearest point. User must do that by hand if desired.

Parameters

points [(N, 2) or (2,) numpy.ndarray] List of cartesian points to convert to polar domain

First column is x and second column is y

Returns

polarPoints [(N, 2) or (2,) numpy.ndarray] Corresponding polar points from cartesian points using *ImageTransform*

See also:

getPolarPointsImage(), getPolarPoints(), getPolarPoints2()

polarTransform.**getCartesianPointsImage**(*points*, *settings*)

Convert list of polar points from image to cartesian image points based on transform metadata

Warning: Cleaner and more succinct to use *ImageTransform.getCartesianPointsImage()*

Note: This does **not** convert from polar to cartesian points, but rather converts pixels from polar image to pixels from cartesian image using *ImageTransform*.

The returned points are not rounded to the nearest point. User must do that by hand if desired.

Parameters

points [(N, 2) or (2,) numpy.ndarray] List of polar points to convert to cartesian domain

First column is r and second column is theta

settings [*ImageTransform*] Contains metadata for conversion from polar to cartesian domain

Settings contains many of the arguments in *convertToPolarImage()* and *convertToCartesianImage()* and provides an easy way of passing these parameters along without having to specify them all again.

Returns

cartesianPoints [(N, 2) or (2,) numpy.ndarray] Corresponding cartesian points from polar `points` using `settings`

See also:

ImageTransform.getCartesianPointsImage(), `getCartesianPoints()`, `getCartesianPoints2()`

CHAPTER 4

Indices and tables

- genindex
- modindex

Python Module Index

p

Index