# xy_python_utils Documentation

***Release 0.1.dev***

**Ying Xiong**

April 14, 2015

Contents

Python utilities by Ying Xiong.

CHAPTER 1

# Getting Started

Install prerequisites:

```
pip install -r requirements.txt
```

Install this package in development mode:

```
python setup.py develop
```

Run unit tests:

```
cd xy_python_utils
python -m unittest discover -p "*_test.py"
cd ..
```

Generate documentation:

```
cd docs
make html
cd ..
```

CHAPTER 2

# Image Utilities

Some utility functions to handle images.

image_utils.**create_icon_mosaic**(*icons*, *icon_shape=None*, *border_size=1*, *border_color=None*, *empty_color=None*, *mosaic_shape=None*, *mosaic_dtype=<Mock name='mock.float' id='140275493000336'>*)

Create a mosaic of image icons.

**Parameters** **icons: a list of icons to be put together for mosaic.**

Currently we require all icons to be multi-channel images of the same size.

**icon_shape: the shape of icons in the output mosaic.**

If not specified, use the shape of first image in *icons*.

**border_size: the size of border.**

**border_color: the color of border, black if not specified.**

**empty_color: the color for empty cells, black if not specified.**

**mosaic_shape: the shape of output mosaic.**

If not specified, try to make a square mosaic according to number of icons.

**mosaic_dtype: the 'dtype' of output mosaic.**

**Returns** The created mosaic image.

image_utils.**imresize**(*img*, *size*)

Resize the input image.

**Parameters** **img: the input image to be resized.**

**size: a scalar for 'scale' or a 2-tuple for '(num_rows, num_cols)'.**

One of the *num_rows* or *num_cols* can be -1, which will be inferred such that the output image has the same aspect ratio as the input.

**Returns** The resized image.

CHAPTER 3

# Matplotlib Utilities

matplotlib_utils.**axes_equal_3d**(*ax=None*)

Mimic Matlab's *axis equal* command. The matplotlib's command *ax.set_aspect("equal")* only works for 2D plots, but not for 3D plots (those generated with *projection="3d"*).

**Parameters** **ax: the axes whose x,y,z axis to be equalized.**

If not specified, default to *plt.gca()*.

matplotlib_utils.**draw_with_fixed_lims**(*ax*, *draw_fcn*)

Save the *xlim* and *ylim* of *ax* before a drawing action, and restore them after the drawing. This is typically useful when one first does an *imshow* and then makes some annotation with *plot*, which will change the limits if not using this function.

matplotlib_utils.**impixelinfo**(*ax=None*, *image=None*)

Mimic Matlab's *impixelinfo* function that shows the image pixel information as the cursor swipes through the figure.

**Parameters** **ax: the axes that tracks cursor movement and prints pixel information.**

We require the *ax.images* list to be non-empty, and if more than one images present in that list, we examine the last (newest) one. If not specified, default to 'plt.gca()'.

**image: if specified, use this 'image''s pixel instead of 'ax.images[-1]''s.**

The replacement *image* must have the same dimension as *ax.images[-1]*, and we will still be using the *extent* of the latter when tracking cursor movement.

**Returns** None

matplotlib_utils.**implay**(*volume*, *fps=20*, *ax=None*)

Play a sequence of image in *volume* as a video.

**Parameters** **volume: the video volume to be played.**

Its size can be either MxNxK (for single-channel image per frame) or MxNxCxK (for multi-channel image per frame).

**fps: the frame rate of the video.**

**ax: the axes in which the video to be played.**

If not specified, default to *plt.gca()*.

matplotlib_utils.**imshow**(*ax*, *img*, *xlim=None*, *ylim=None*, ***kw*)

Enhance *ax.imshow* with coordinate limits.

**Parameters** **ax: the axes in which an image will be drawn.**

**img: the 2D image to be drawn.**

**xlim, ylim: the horizontal coordinate limits of the image.**

This will set the *extent* parameter of *ax.imshow*, which is relatively inconvenient to set directly because of the half-pixel issue. Default: (0, *num_cols*-1), (0, *num_rows*-1).

****kw: other parameters to be passed to 'ax.imshow'.**

The *extent* will be ignored if presented.

**Returns** None

matplotlib_utils.**tight_subplot**(*num_rows*, *num_cols*, *plot_index*, *gap=0.01*, *marg_h=0.01*, *marg_w=0.01*, *fig=None*)

Add a tight subplot axis to the current (or a given) figure.

**Parameters num_rows: number of rows.**

**num_cols: number of columns.**

**plot_index: the index to the subplot.**

**gap: the gap between axes, scalar or 2-tuple '(gap_h, gap_w)'.**

Value should be between (0, 1).

**marg_h: the margins in height, scalar or 2-tuple '(lower, upper)'.**

Value should be between (0, 1).

**marg_w: the margins in width, scalar or 2-tuple '(left, right)'.**

Value should be between (0, 1).

**fig: figure to which the new axes to be added to**

Default to *plt.gcf()* if not specified.

**Returns** The newly added axes.

CHAPTER 4

# Numerical Differentiation

numerical_differentiation.**numerical_jacobian**(*fcn*, *x0*, *dx=1e-06*, *method=0*, *return_f0=False*)

Compute the numerical Jacobian matrix of a given function.

**Parameters** **fcn: a function handle that takes an N-vector as input and return an M-vector.**

**x0: an input N-vector.**

**dx: a scalar for small change in x0.**

**method: a integer or string with following options:**

- {0, 'forward'}: compute the Jacobian as (f(x0+dx)-f(x0))/dx.
- 1, 'central' : compute the Jacobian as (f(x0+dx)-f(x0-dx))/2/dx.

**return_f0: if set to true, also return fcn(x0).**

**Returns** J: the MxN Jacobian matrix.

f0: the function value at x0.

### Examples

```
>>> J = numerical_jacobian(fcn, x0, ...)
>>> (J, f0) = numerical_jacobian(fcn, x0, ..., return_f0=True)
```

CHAPTER 5

# Numpy Utilities

Some extended utility functions for 'numpy' module.

numpy_utils.**meshgrid_nd**(**arrs*)

Multi-dimensional meshgrid.

**Parameters** **x, y, z, ...: ndarray**

Multiple 1-D arrays representing the coordinates of the grid.

**Returns** X, Y, Z, ...: ndarray

Multi-dimensional arrays of shape (len(x), len(y), len(z), ...). Note that there is a discrepancy to the original 2D meshgrid, where the output array shape is swapped, i.e. (len(y), len(x)). Specifically, if:

```
X, Y = meshgrid(x, y)
X2, Y2 = meshgrid_nd(x, y)
```

then we have *X* == *X2.T* and *Y* == *Y2.T*.

### Examples

```
>>> X, Y, Z = np.meshgrid([1,2,3], [10,20], [-2,-3,-4,-5])
>>> X
array([[[1, 1, 1, 1],
        [1, 1, 1, 1]],
       [[2, 2, 2, 2],
        [2, 2, 2, 2]],
       [[3, 3, 3, 3],
        [3, 3, 3, 3]]])
>>> Y
array([[[10, 10, 10, 10],
        [20, 20, 20, 20]],
       [[10, 10, 10, 10],
        [20, 20, 20, 20]],
       [[10, 10, 10, 10],
        [20, 20, 20, 20]]])
>>> Z
array([[[-2, -3, -4, -5],
        [-2, -3, -4, -5]],
       [[-2, -3, -4, -5],
        [-2, -3, -4, -5]],
```

```
            [[-2, -3, -4, -5],
             [-2, -3, -4, -5]]])
```

numpy_utils.**null**(*A*, *tol=1e-12*)

Return the null space of matrix or vector *A*, such that:

```
dot(A, null(A)) == eps(M, N)
```

Each column *r* of *null(A)* is a unit vector, and ||*dot(A, r)*|| < *tol*.

CHAPTER 6

# OS Utilities

Some extended utility functions for 'os' module.

os_utils.**cp_r**(*src*, *dst*)

Same effect as the unix command 'cp -r src dst', supporting the followings:

1. *cp_r("/path/to/src_file", "/path/to/"dst_file")*: The 'src_file' is a single file, and 'dst_file' is created or overwritten if already exists.
2. *cp_r("/path/to/src_folder", "/path/to/dst_folder")*: The 'dst_folder' is a single folder, and 'dst_folder' will be created if not already exists, otherwise a "/path/to/dst_folder/src_folder" will be created.
3. *cp_r("/path/to/src", "/path/to/dst_folder")*: The 'src' can be either a file or a folder, and can contain wildcard characters (e.g. '*'), and the 'dst_folder' must already exist.
4. *cp_r(["/path/to/src1", "/path/to/src2", ...], "/path/to/dst_folder")*: The 'src' can be anything as the previous syntax, and the first argument can be either list or tuple. The 'dst_folder' must already exist.

os_utils.**mkdir_p**(*path*[, *mode=0o777* ])

Create a leaf directory 'path' and all intermediate ones, and no error will be reported if the directory already exists. Same effect as the unix command 'mkdir -p path'.

os_utils.**rm_rf**(*path*)

Remove a file or a directory, recursively. No error will be reported if 'path' does not exist. The 'path' can be a list or tuple. Same effect as the unix command 'rm -rf path'.

CHAPTER 7

# Quaternion

CHAPTER 8

# Unittest Utilities

Utility functions for unit test.

unittest_utils.**check_gradient**(*fcn*, *dfcn*, *N*, *x0=None*, *dx=None*, *delta=0.0001*, *m=0.01*, *M=10*, *raise_exception=True*)

Numerically check whether 'dfcn' calculates the gradient of 'fcn'.

More specifically, this function checks whether the following quantities are close to each other

- f(x) - f(x0)
- (x-x0) cdot f'(x0)

We consider them to be close enough if **either one** of the following is true

1. the absolute difference is smaller than (m * ||x-x0||);
2. the relative difference is smaller than (M * ||x-x0||).

**Parameters** **fcn: a function handler with a single (vector or scalar) input and a**

scalar output.

**dfcn: a function handler with a single (vector or scalar) input and a**

vector output for gradient of 'fcn'. [NOTE]: Another option is to let dfcn=None (or something else that is not callable, e.g. [] or True), and fcn return a 2-tuple for both fucntion value and its gradient.

**N: the dimensionality of input to the fucntion, which is a Nx1 vector.**

**x0: the initial input point evaluated by the function, with default**

{randn(N)}.

**dx, delta: the direction of evaluation point moves, such that::**

x = x0 + delta*dx

with 'dx' a unit Nx1 vector and 'delta' a scalar.

**m, M: the thresholds described above.**

unittest_utils.**check_jacobian**(*fcn*, *dfcn*, *N*, *x0=None*, *dx=None*, *delta=0.0001*, *m=0.01*, *M=10*, *raise_exception=True*)

Numerically check whether 'dfcn' calculates the Jacobian of 'fcn'.

More specifically, whether the following vectors are close to each other

- f(x) - f(x0)

- J(x0) cdot (x-x0)

We consider them to be close enough if **either one** of the following is true

1. "absolutely" close with tolerance m*||x-x0|| (see 'check_near_abs');
2. "relatively" close with tolerance M*||x-x0|| (see 'check_near_rel').

**Parameters** **fcn: a function handler with a single (vector or scalar) input and a**

vector output.

**dfcn: a function handler with a single (vector or scalar) input and a**

matrix output for Jacobian of 'fcn'. [NOTE]: Another option is to let dfcn=None (or something else that is not callable, e.g. [] or True), and fcn return a 2-tuple for both fucntion value and its Jacobian.

**The rest is the same as "check_gradient".**

unittest_utils.**check_near**(*v1*, *v2*, *tol*, *raise_exception=True*)

Check whether scalar/vector/matrix 'v1' and 'v2' are close to each other under tolerance tol, in the sense that:

```
(absolute)   ||v1 - v2|| <= tol,   **or**
(relative)   ||v1 - v2|| / max(||v1||, ||v2||, eps) <= tol,
```

where ||.|| is the Frobenius norm.

unittest_utils.**check_near_abs**(*v1*, *v2*, *tol*, *raise_exception=True*)

Same as 'check_near' but only check in the "absolute" sense.

unittest_utils.**check_near_rel**(*v1*, *v2*, *tol*, *raise_exception=True*)

Same as 'check_near' but only check in the "relative" sense.

CHAPTER 9

# General Utilities

Some general utility classes and functions.

**class** `utils.`**`Range`**(*start*, *stop=None*, *step=None*)

A range of numbers from *start* (inclusive) to *end* (exclusive) with a given *step*. This class is similar to the *range* built-in in python3, but also supports floating point parameters.

Note the rounding effect when using floating point parameters. The suggested way is to pad an *epsilon* at the stop point:

```
Range(1.5, 1.8001, 0.3)    # 1.8 will be included.
Range(1.5, 1.7999, 0.3)    # 1.5 will be excluded.
Range(1.5, 1.8, 0.3)       # 1.8 should be excluded, but might not be
                           # because of rounding effect. Avoid this.
```

CHAPTER 10

# Indices and tables

- *genindex*
- *modindex*
- *search*

## i



## m



## n



## o



## u