xy_python_utils Documentation

Release 0.1.dev

Ying Xiong

June 02, 2017

Contents

Python utilities by Ying Xiong.

CHAPTER 1

Getting Started

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

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: axes, optional**

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*)

Perform plot without changing the *xlims* and *ylims* of the axes.

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: axes**

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: ndarray

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*, ***kw*)

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

Parameters **volume: ndarray**

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: int, optional

The frame rate of the video.

ax: axes, optional

The axes in which the video to be played. If not specified, default to *plt.gca()*.

****kw: key-value pairs**

Other parameters to be passed to *ax.imshow*, e.g. *cmap="gray"*, *vmin=0*, *vmax=1*, etc.

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

Enhance *ax.imshow* with coordinate limits.

Parameters ax: axes

The axes in which an image will be drawn.

img: ndarray

The 2D image to be drawn.

xlim, ylim: 2-tuple, optional

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: key-value pairs**

Other parameters to be passed to *ax.imshow*. The *extent* will be ignored if presented.

Returns The *AxesImage* returned by *ax.imshow*.

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, num_cols: int

Number of rows / columns.

plot_index: int

The index to the subplot.

gap: float between (0,1), optional

The gap between axes, scalar or 2-tuple *(gap_h, gap_w)*.

marg_h: float between (0,1), optional

The margins in height, scalar or 2-tuple *(lower, upper)*.

marg_w: float between (0,1), optional

The margins in width, scalar or 2-tuple *(left, right)*.

fig: Figure, optional

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: function handle**

Takes an N-vector as input and return an M-vector.

x0: ndarray

An input N-vector.

dx: scalar

For small change in x0.

method: int 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: boolean

If set to true, also return fcn(x0).

Returns **J** : *ndarray*

The *MxN* Jacobian matrix.

f0 : *ndarray*

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=511*)

Create a leaf directory 'path' and all intermediate ones.

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'.

Quaternion

Definition

A quaternion $\mathbf{q}$ is represented as a 4-tuple (a, b, c, d), with basis $\{1, i, j, k\}$ written as

$$\mathbf{q} = (a, b, c, d) = a + b\, i + c\, j + d\, k. \tag{7.1}$$

The basis elements have multiplication property

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1, \\ ij = k,\ jk = i,\ ki = j, \\ ji = -k,\ kj = -i,\ ik = -j. \end{aligned}$$

The *Hamilton product* of two general quaternion is

$$\begin{aligned} &(a_1, b_1, c_1, d_1)(a_2, b_2, c_2, d_2) \\ &= (a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2, \\ &\quad a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2, \\ &\quad a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2, \\ &\quad a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2). \end{aligned} \tag{7.2}$$

A quaternion can be divided into a *scalar part* and a *vector part*

$$\mathbf{q} = (r, \mathbf{v}), \quad \text{with } r \in \mathbb{R}, \mathbf{v} \in \mathbb{R}^3.$$

We also consider scalar r and 3-vector $\mathbf{v}$ as special forms of quaternion

$$\mathbf{q}_r = (r, \mathbf{0}), \quad \mathbf{q}_\mathbf{v} = (0, \mathbf{v}),$$

and write $\mathbf{q}_r$ and r ($\mathbf{q}_\mathbf{v}$ and $\mathbf{v}$) interchangably in this note.

For quaternion $\mathbf{q}$ defined in (7.1), its *conjugate* is

$$\mathbf{q}^* = a - b\, i - c\, j - d\, k,$$

its *norm* is

$$\|\mathbf{q}\| = \sqrt{\mathbf{q}\mathbf{q}^*} = \sqrt{\mathbf{q}^*\mathbf{q}} = \sqrt{a^2 + b^2 + c^2 + d^2}, \tag{7.3}$$

and its *reciprocal* is

$$\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{\|\mathbf{q}\|^2}, \quad \mathbf{q}\mathbf{q}^{-1} = \mathbf{q}^{-1}\mathbf{q} = 1. \tag{7.4}$$

Note that the multiplications in (7.3) and (7.4) are Hamilton product defined in (7.2).

Spatial Rotation

Given a unit vector $\widehat{\mathbf{u}} = (u_x, u_y, u_z)$ with a scalar angle θ, we define quaternion

$$\mathbf{q} = \exp\left(\frac{\theta}{2}(u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\right) = \cos\left(\frac{\theta}{2}\right) + \sin\left(\frac{\theta}{2}(u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\right)$$

then for any given vector $\mathbf{p}$, its rotation across axis $\widehat{\mathbf{u}}$ for angle θ is

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1},$$

using Hamilton product (7.2). Note that both $\mathbf{q}$ and $-\mathbf{q}$ performs the same rotation.

Conversion between rotation matrics

Given a unit quaternion $\mathbf{q} = (a, b, c, d)$, it can be converted to a rotation matrix as

$$\boldsymbol{R} = \begin{bmatrix} 1 - 2c^2 - 2d^2 & 2bc - 2ad & 2bd + 2ac \\ 2bc + 2ad & 1 - 2b^2 - 2d^2 & 2cd - 2ab \\ 2bd - 2ac & 2cd + 2ab & 1 - 2b^2 - 2c^2 \end{bmatrix}$$

To convert from a rotation matrix $\boldsymbol{R}$ to a quaternion,

$$\begin{aligned} \mathbf{q} = \Big(&\frac{1}{2}\sqrt{R_{11} + R_{22} + R_{33} + 1}, \\ &\frac{1}{2}\sqrt{R_{11} - R_{22} - R_{33} + 1}\,\text{sign}(R_{32} - R_{23}), \\ &\frac{1}{2}\sqrt{-R_{11} + R_{22} - R_{33} + 1}\,\text{sign}(R_{13} - R_{31}), \\ &\frac{1}{2}\sqrt{-R_{11} - R_{22} + R_{33} + 1}\,\text{sign}(R_{21} - R_{12})\Big). \end{aligned}$$

This conversion can be implemented with a single square root, but one needs to take special care on numerical stability when doing so.

API References

Utility functions for quaternion and spatial rotation.

A quaternion is represented by a 4-vector q as:

```
q = q[0] + q[1]*i + q[2]*j + q[3]*k.
```

The validity of input to the utility functions are not explicitly checked for efficiency reasons.

Abbr.	Meaning
quat	Quaternion, 4-vector.
vec	Vector, 3-vector.
ax, axis	Axis, 3- unit vector.
ang	Angle, in unit of radian.
rot	Rotation.
rotMatx	Rotation matrix, 3x3 orthogonal matrix.
HProd	Hamilton product.
conj	Conjugate.
recip	Reciprocal.

quaternion.**quatConj**(*q*)
Return the conjugate of quaternion *q*.

quaternion.**quatFromAxisAng**(*ax*, *theta*)
Get a quaternion that performs the rotation around axis *ax* for angle *theta*, given as:

```
q = (r, v) = (cos(theta/2), sin(theta/2)*ax).
```

Note that the input *ax* needs to be a 3x1 unit vector.

quaternion.**quatFromRotMatx**(*R*)
Get a quaternion from a given rotation matrix *R*.

quaternion.**quatHProd**(*p*, *q*)
Compute the Hamilton product of quaternions *p* and *q*.

quaternion.**quatRecip**(*q*)
Compute the reciprocal of quaternion *q*.

quaternion.**quatToRotMatx**(*q*)
Get a rotation matrix from the given unit quaternion *q*.

quaternion.**rotVecByAxisAng**(*u*, *ax*, *theta*)
Rotate the 3-vector *u* around axis *ax* for angle *theta* (radians), counter-clockwisely when looking at inverse axis direction. Note that the input *ax* needs to be a 3x1 unit vector.

quaternion.**rotVecByQuat**(*u*, *q*)
Rotate a 3-vector *u* according to the quaternion *q*. The output *v* is also a 3-vector such that:

```
[0; v] = q * [0; u] * q^{-1}
```

with Hamilton product.

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: function handler

Takes a single (vector or scalar) as input and outputs a scalar.

dfcn: function handler

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

N: int

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: float, optional

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: function handler**

Takes a single (vector or scalar) as input and outputs a vector.

dfcn: function handler

Takes a single (vector or scalar) as input and outputs a matrix for Jacobian of *fcn*. NOTE: Another option is to let dfcn=None (or something else that is not callable, e.g. *[]*), 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*

m

n

o

q

u