# Pyimof

*Release 1.0.0*

**R. Fezzani**

**Aug 13, 2019**

# CONTENTS:

CHAPTER

ONE

# OVERVIEW

Pyimof actually only supports gray scale images in its implemented methods.

## 1.1 Implemented methods

Optical flow algorithms can mainly be classified into

- **global methods** based on pixel-wise matching costs with regularization constraints, *ie* based on the Horn & Schunck paradigm.
- **local methods** based on local window matching costs, *ie* based on the Lucas-Kanade paradigm.

Pyimof provides one implementation for each of these class of methods:

- **TV-L1**: A popular algorithm introduced by Zack *et al.*[1], improved in[2] and detailed in[3] (See `pyimof.solvers.tvl1()`).
- **Iterative Lucas-kanade**: A fast and robust algorithm developped by Le Besnerais and Champagnat[4] and improved in[5] (See `pyimof.solvers.ilk()`).

These two algorithms have been selected for theire relative speed. Efficient GPU implementations for both of them have been developped but it is not planned to port Pyimof on this platform.

## 1.2 Datasets

Multiple datasets for the evaluation of optical flow algorithms have been developped (for example Middlebury, MPI-Sintel and Flying Chairs). The two-frame grayscale **Middlebury** training dataset[6] is accessible via the `pyimof.data` module functions to ease testing.

[1] Zach, C., Pock, T., & Bischof, H. (2007, September). A duality based approach for realtime TV-L 1 optical flow. In Joint pattern recognition symposium (pp. 214-223). Springer, Berlin, Heidelberg.

[2] Wedel, A., Pock, T., Zach, C., Bischof, H., & Cremers, D. (2009). An improved algorithm for TV-L 1 optical flow. In Statistical and geometrical approaches to visual motion analysis (pp. 23-45). Springer, Berlin, Heidelberg.

[3] Pérez, J. S., Meinhardt-Llopis, E., & Facciolo, G. (2013). TV-L1 optical flow estimation. Image Processing On Line, 2013, 137-150.

[4] Le Besnerais, G., & Champagnat, F. (2005, September). Dense optical flow by iterative local window registration. In IEEE International Conference on Image Processing 2005 (Vol. 1, pp. I-137). IEEE.

[5] Plyer, A., Le Besnerais, G., & Champagnat, F. (2016). Massively parallel Lucas Kanade optical flow for real-time video processing applications. Journal of Real-Time Image Processing, 11(4), 713-730.

[6] Baker, S., Scharstein, D., Lewis, J. P., Roth, S., Black, M. J., & Szeliski, R. (2011). A database and evaluation methodology for optical flow. International Journal of Computer Vision, 92(1), 1-31.

## 1.3 IO

Estimated vector fields can be saved and loaded in the *.flo* file format using the `pyimof.io` module.

## 1.4 Visualization

Visualizing optical flow is made easy using the `pyimof.display` module

- the `pyimof.display.plot()` function applies a colormap (preferably circular ) to the optical flow according to its direction and magnitude. An optional color-wheel showing the color code can also be displayed to ease resulting image understanding.
- the `pyimof.display.quiver()` function draws a quiver plot with multiple option for coloring the arrows and displaying a background image.

Moreover, Pyimof gives access to a Matplotlib colormap inspired by the popular color code used by the Middlebury evaluation site for displaying algorithms' results.

### 1.4.1 References

CHAPTER

TWO

# PYIMOF

## 2.1 pyimof package

### 2.1.1 Subpackages

#### pyimof.data package

#### Module contents

This module gives access to the two-frames gray scale training sequences of the Middlebury optical flow dataset[1].

`pyimof.data.`**`beanbags`**(*, *seqname='Beanbags'*)

`pyimof.data.`**`dimetrodon`**(*, *seqname='Dimetrodon'*)

`pyimof.data.`**`dogdance`**(*, *seqname='DogDance'*)

`pyimof.data.`**`grove2`**(*, *seqname='Grove2'*)

`pyimof.data.`**`grove3`**(*, *seqname='Grove3'*)

`pyimof.data.`**`hydrangea`**(*, *seqname='Hydrangea'*)

`pyimof.data.`**`minicooper`**(*, *seqname='MiniCooper'*)

`pyimof.data.`**`rubberwhale`**(*, *seqname='RubberWhale'*)

`pyimof.data.`**`urban2`**(*, *seqname='Urban2'*)

`pyimof.data.`**`urban3`**(*, *seqname='Urban3'*)

`pyimof.data.`**`venus`**(*, *seqname='Venus'*)

`pyimof.data.`**`walking`**(*, *seqname='Walking'*)

### 2.1.2 Submodules

### 2.1.3 pyimof.display module

Collection of utils and functions for the visualization of vector fields.

[1] Baker, S., Scharstein, D., Lewis, J. P., Roth, S., Black, M. J., & Szeliski, R. (2011). A database and evaluation methodology for optical flow. International Journal of Computer Vision, 92(1), 1-31.

`pyimof.display.color_wheel`(*u=None*, *v=None*, *nr=50*, *ntheta=1025*)

Compute the discretization of a wheel used to describe the color code used to display a vector field (u, v).

If the vector field (u, v) is provided, the radius of the wheel is equal to its maximum magnitude. Otherwise (i.e. if any of u and v is None), the radius is set to 1.

**Parameters**

- **u** (*ndarray (optional)*) – The horizontal component of the vector field (default: None).
- **v** (*ndarray (optional)*) – The vertical component of the vector field (default: None).
- **nr** (*int (optional)*) – The number of steps used to discretise the wheel radius.
- **ntheta** (*int (optional)*) – The number of steps used to discretise the wheel sectors.

**Returns** **angle, radius** – The grid discretisation of the wheel sectors and radius.

**Return type** tuple[ndarray]

`pyimof.display.flow_to_color`(*u*, *v*, *cmap=None*, *scale=True*)

Apply color code to a vector field according to its orientation and magnitude.

Any colormap compatible with matplotlib can be applyed but circular colormaps are recommanded ( for example 'huv', 'twilight', 'twilight_shifted' and the builtin 'middlebury' colormaps).

If cmap is None, the HSV image defined using optical flow orientation (hue) and magnitude (saturation) is returned.

**Parameters**

- **u** (*ndarray*) – The horizontal component of the vector field.
- **v** (*ndarray*) – The vertical component of the vector field.
- **cmap** (*str (optional)*) – The colormap used to color code the input vector field.
- **scale** (*bool (optional)*) – whether to scale output saturation according to magnitude.

**Returns** **img** – RGBA image representing the desired color code applyed to the vector field.

**Return type** ndarray

`pyimof.display.get_tight_figsize`(*I*)

Computes the matplotlib figure tight size respecting image proportions.

**Parameters** **I** (*ndarray*) – The image to be displayed.

**Returns** **w, h** – The width and height in inch of the desired figure.

**Return type** tuple[float]

`pyimof.display.plot`(*u*, *v*, *ax=None*, *cmap='middlebury'*, *scale=True*, *colorwheel=True*)

Plots the color coded vector field.

**Parameters**

- **u** (*ndarray*) – The horizontal component of the vector field.
- **v** (*ndarray*) – The vertical component of the vector field.
- **ax** (*Axes (optional)*) – Optional matplotlib axes used to plot the image. If None, the image is displayed in a tight figure.
- **cmap** (*str (optional)*) – The colormap used to color code the input vector field.

- **scale** (*bool (optional)*) – whether to scale output saturation according to magnitude.
- **colorwheel** (*bool (optional)*) – whether to display the color wheel describing the images colors or not.

**Returns** **ax** – The matplotlib axes where the image is displayed.

**Return type** Axes

`pyimof.display.quiver`(*u, v, c=None, bg=None, ax=None, step=None, nvec=50, bg_cmap=None, **kwargs*)

Draws a quiver plot representing a dense vector field.

**Parameters**

- **u** (*ndarray (with shape m×n)*) – The horizontal component of the vector field.
- **v** (*ndarray (with shape m×n)*) – The vertical component of the vector field.
- **c** (*ndarray (optional (with shape m×n))*) – Values used to color the arrows.
- **bg** (*ndarray (2D or 3D optional)*) – Background image.
- **ax** (*Axes (optional)*) – Axes used to plot the image. If None, the image is displayed in a tight figure.
- **step** (*int (optional)*) – The grid step used to display the vector field. If None, it is computed using the nvec parameter.
- **nvec** (*int*) – The maximum number of vector over all the grid dimentions. It is ignored if the step parameter is not None.
- **bg_cmap** (*str (optional)*) – The colormap used to color the background image.

#### Notes

Any other `matplotlib.pyplot.quiver()` valid keyword can be used, knowing that some are fixed

- units = 'dots'
- angles = 'xy'
- scale = 'xy'

**Returns** **ax** – The matplotlib axes where the vector field is displayed.

**Return type** Axes

### 2.1.4 pyimof.io module

Functions to read and write '.flo' Middleburry file format.

`pyimof.io.floread`(*fname*)

Read a Middleburry .flo file.

**Parameters** **fname** (*str*) – The file name.

**Returns** **u, v** – The horizontal and vertical components of the estimated optical flow.

**Return type** tuple[ndarray]

`pyimof.io.flowrite`(*u, v, fname*)

Write a given flow to the Middleburry file format .flo

**Parameters**

- **u** (*ndarray*) – The horizontal component of the estimated optical flow.
- **v** (*ndarray*) – The vertical component of the estimated optical flow.
- **fname** (*str*) – The target file name. The '.flo' extension is appended if necessary.

### 2.1.5 pyimof.solvers module

Collection of optical flow algorithms.

`pyimof.solvers.ilk`(*I0*, *I1*, *rad=7*, *nwarp=10*, *gaussian=True*, *prefilter=False*)

Coarse to fine iterative Lucas-Kanade (iLK) optical flow estimator. A fast and robust algorithm developped by Le Besnerais and Champagnat[4] and improved in[5]..

**Parameters**

- **I0** (*ndarray*) – The first gray scale image of the sequence.
- **I1** (*ndarray*) – The second gray scale image of the sequence.
- **rad** (*int*) – Radius of the window considered around each pixel.
- **nwarp** (*int*) – Number of times I1 is warped.
- **gaussian** (*bool*) – if True, gaussian kernel is used otherwise uniform kernel is used.
- **prefilter** (*bool*) – whether to prefilter the estimated optical flow before each image warp.

**Returns** **u, v** – The horizontal and vertical components of the estimated optical flow.

**Return type** tuple[ndarray]

#### References

#### Examples

```
>>> from matplotlib import pyplot as plt
>>> import pyimof
>>> I0, I1 = pyimof.data.yosemite()
>>> u, v = pyimof.solvers.ilk(I0, I1)
>>> pyimof.display.plot(u, v)
>>> plt.show()
```

`pyimof.solvers.tvl1`(*I0*, *I1*, *dt=0.2*, *lambda_=15*, *tau=0.3*, *nwarp=5*, *niter=10*, *tol=0.0001*, *prefilter=False*)

Coarse to fine TV-L1 optical flow estimator. A popular algorithm intrudced by Zack et al. [1]_, improved in[2] and detailed in[3].

**Parameters**

- **I0** (*ndarray*) – The first gray scale image of the sequence.

[4] Le Besnerais, G., & Champagnat, F. (2005, September). Dense optical flow by iterative local window registration. In IEEE International Conference on Image Processing 2005 (Vol. 1, pp. I-137). IEEE.

[5] Plyer, A., Le Besnerais, G., & Champagnat, F. (2016). Massively parallel Lucas Kanade optical flow for real-time video processing applications. Journal of Real-Time Image Processing, 11(4), 713-730.

[2] Wedel, A., Pock, T., Zach, C., Bischof, H., & Cremers, D. (2009). An improved algorithm for TV-L 1 optical flow. In Statistical and geometrical approaches to visual motion analysis (pp. 23-45). Springer, Berlin, Heidelberg.

[3] Pérez, J. S., Meinhardt-Llopis, E., & Facciolo, G. (2013). TV-L1 optical flow estimation. Image Processing On Line, 2013, 137-150.

- **I1** (*ndarray*) – The second gray scale image of the sequence.
- **dt** (*float*) – Time step of the numerical scheme. Convergence is proved for values dt < 0.125, but it can be larger for faster convergence.
- **lambda_** (*float*) – Attachement parameter. The smaller this parameter is, the smoother is the solutions.
- **tau** (*float*) – Tightness parameter. It should have a small value in order to maintain attachement and regularization parts in correspondence.
- **nwarp** (*int*) – Number of times I1 is warped.
- **niter** (*int*) – Number of fixed point iteration.
- **tol** (*float*) – Tolerance used as stopping criterion based on the $L^2$ distance between two consecutive values of (u, v).
- **prefilter** (*bool*) – whether to prefilter the estimated optical flow before each image warp.

**Returns** **u, v** – The horizontal and vertical components of the estimated optical flow.

**Return type** tuple[ndarray]

#### References

#### Examples

```
>>> from matplotlib import pyplot as plt
>>> import pyimof
>>> I0, I1 = pyimof.data.yosemite()
>>> u, v = pyimof.solvers.tvl1(I0, I1)
>>> pyimof.display.plot(u, v)
>>> plt.show()
```

### 2.1.6 pyimof.util module

Common tools to optical flow algorithms.

`pyimof.util.`**`coarse_to_fine`**(*I0*, *I1*, *solver*, *downscale=2*, *nlevel=10*, *min_size=16*)

Generic coarse to fine solver.

**Parameters**

- **I0** (*ndarray*) – The first gray scale image of the sequence.
- **I1** (*ndarray*) – The second gray scale image of the sequence.
- **solver** (*callable*) – The solver applyed at each pyramid level.
- **downscale** (*float*) – The pyramid downscale factor.
- **nlevel** (*int*) – The maximum number of pyramid levels.
- **min_size** (*int*) – The minimum size for any dimension of the pyramid levels.

**Returns** **u, v** – The horizontal and vertical components of the estimated optical flow.

**Return type** tuple[ndarray]

`pyimof.util.get_pyramid`(*I*, *downscale=2.0*, *nlevel=10*, *min_size=16*)

Construct image pyramid.

**Parameters**

- **I** (*ndarray*) – The image to be preprocessed (Gray scale or RGB).
- **downscale** (*float*) – The pyramid downscale factor.
- **nlevel** (*int*) – The maximum number of pyramid levels.
- **min_size** (*int*) – The minimum size for any dimension of the pyramid levels.

**Returns** **pyramid** – The coarse to fine images pyramid.

**Return type** list[ndarray]

`pyimof.util.resize_flow`(*u*, *v*, *shape*)

Rescale the values of the vector field (u, v) to the desired shape.

The values of the output vector field are scaled to the new resolution.

**Parameters**

- **u** (*ndarray*) – The horizontal component of the motion field.
- **v** (*ndarray*) – The vertical component of the motion field.
- **shape** (*iterable*) – Couple of integers representing the output shape.

**Returns** **ru, rv** – The resized and rescaled horizontal and vertical components of the motion field.

**Return type** tuple[ndarray]

`pyimof.util.tv_regularize`(*x*, *tau=0.3*, *dt=0.2*, *max_iter=100*, *p=None*, *g=None*)

Toltal variation regularization using Chambolle algorithm [1]_.

**Parameters**

- **x** (*ndarray*) – The target array.
- **tau** (*float*) – Tightness parameter. It should have a small value in order to maintain attachement and regularization parts in correspondence.
- **dt** (*float*) – Time step of the numerical scheme. Convergence is proved for values dt < 0.125, but it can be larger for faster convergence.
- **max_iter** (*int*) – Maximum number of iteration.
- **p** (*ndarray*) – Optional buffer array of shape (x.ndim, ) + x.shape.
- **g** (*ndarray*) – Optional buffer array of shape (x.ndim, ) + x.shape.

#### References

### 2.1.7 Module contents

A python package for optical flow estimation and visualization.

CHAPTER

THREE

# PYIMOF GALLERY

**Note:** Click *here* to download the full example code

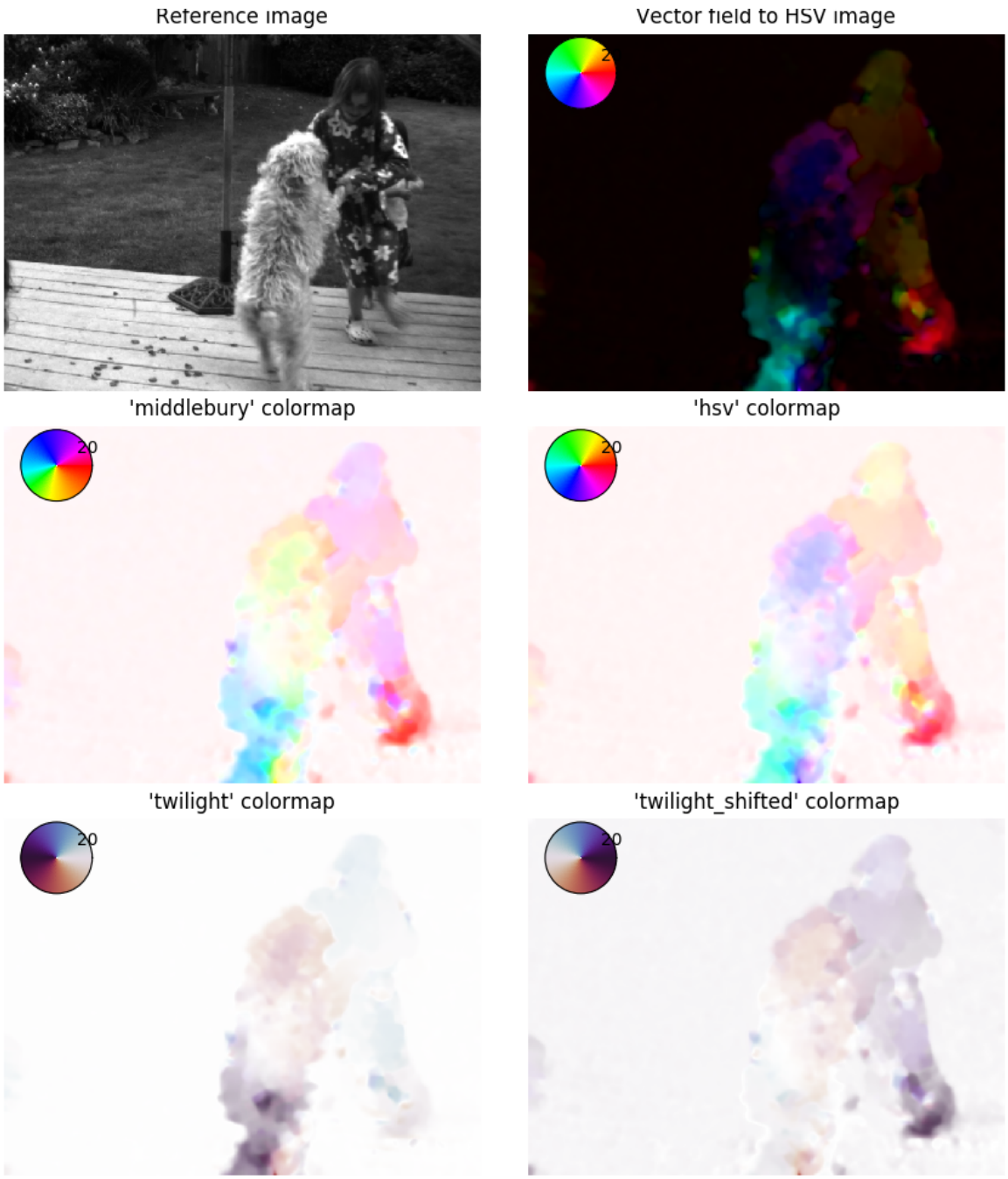
## 3.1 Vector field color coding

Demonstration of colormap application to a vector field.

Pyimof provides the `pyimof.display.plot()` that displays a color mapping applyed to a dense vector field according to its orientation and magnitude. Any circular colormap can be applyed. Matplotlib provides some of them by default: `hsv`, `twilight`, `twilight_shifted`, `hsv_r`, `twilight_r`, `twilight_shifted_r`.

If no colormap is provided to the `pyimof.display.plot()` function, the fector field color coding is made by constructing a HSV image in which the hue is the orientation of the vector flow and the value is its magnitude. The saturation is set to 1.

Pyimof defines the `middlebury` matplotlib colormap that is inspired by the color coding introduced by the Middlebury optical flow evaluation website for displaying algorithms results. Its reverse version `middlebury_r` is also provided.




```
import matplotlib.pyplot as plt
import pyimof

# --- Load the Hydrangea sequence

I0, I1 = pyimof.data.dogdance()

# --- Estimate the optical flow
```

(continues on next page)

(continued from previous page)

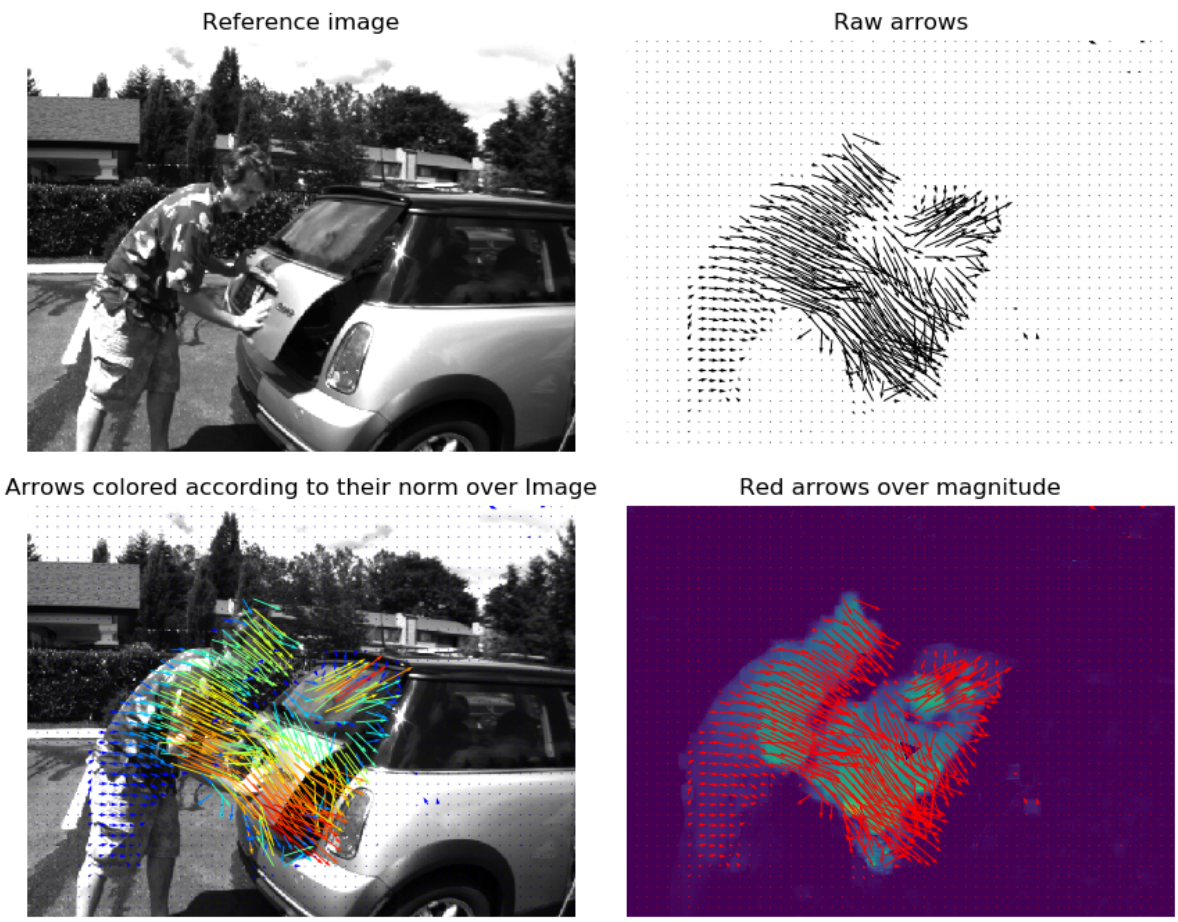
```
u, v = pyimof.solvers.ilk(I0, I1)

# --- Display it with different colormaps

fig = plt.figure(figsize=((9, 10)))
ax_arr = fig.subplots(3, 2, True, True)
fig.tight_layout()

ax0, ax1 = ax_arr[0, :]

ax0.imshow(I0, cmap='gray')
ax0.set_axis_off()
ax0.set_title("Reference image")

pyimof.display.plot(u, v, ax=ax1, cmap=None)
ax1.set_title("Vector field to HSV image")

cmap_list = ['middlebury', 'hsv', 'twilight', 'twilight_shifted']

for ax, cm in zip(ax_arr[1:, :].ravel(), cmap_list):
    pyimof.display.plot(u, v, ax=ax, cmap=cm)
    ax.set_title(f"'{cm}' colormap")

plt.show()
```

**Total running time of the script:** ( 0 minutes 3.931 seconds)

**Note:** Click *here* to download the full example code

## 3.2 Vector field quiver plot

Demonstration of vector field quiver plot.




```
import numpy as np
import matplotlib.pyplot as plt
import pyimof

# --- Load the MiniCooper sequence

I0, I1 = pyimof.data.minicooper()

# --- Estimate the optical flow

u, v = pyimof.solvers.ilk(I0, I1)

norm = np.sqrt(u*u + v*v)

# --- Display it with different options

fig = plt.figure(figsize=((9, 7)))
ax0, ax1, ax2, ax3 = fig.subplots(2, 2, True, True).ravel()
fig.tight_layout()

ax0.imshow(I0, cmap='gray')
ax0.set_axis_off()
ax0.set_title("Reference image")
```

(continues on next page)

(continued from previous page)

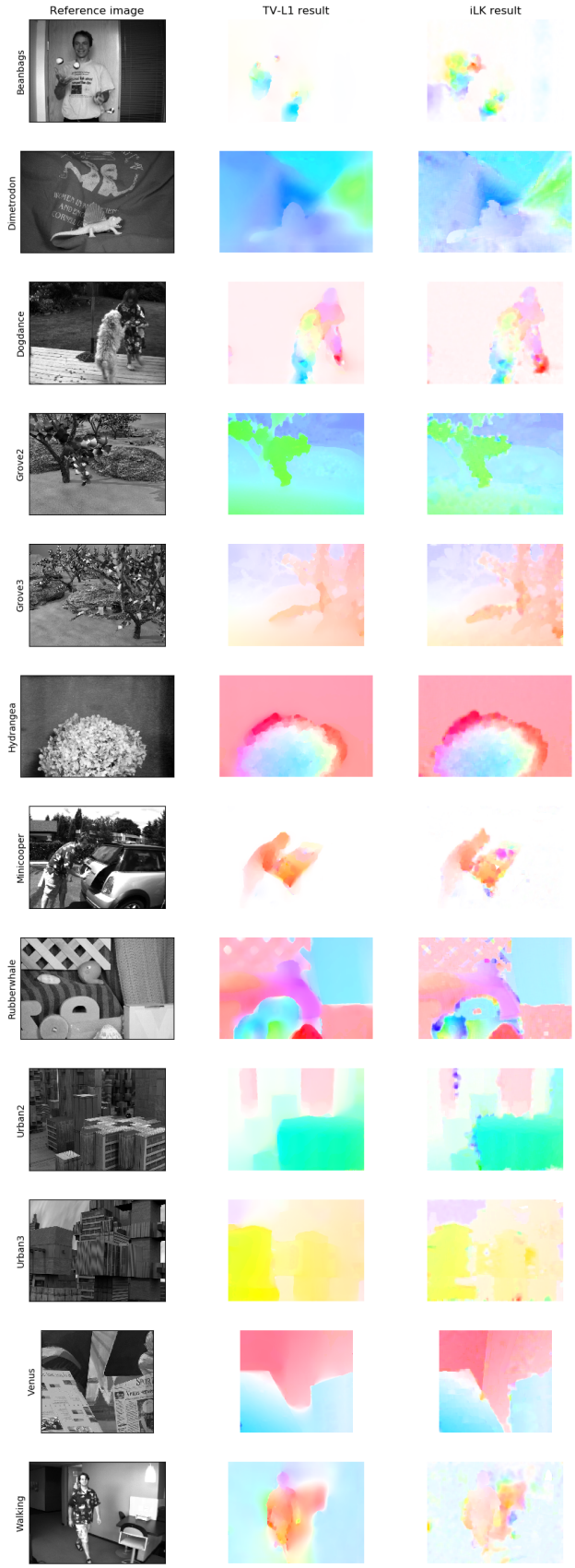
```
pyimof.display.quiver(u, v, ax=ax1)
ax1.set_title("Raw arrows")

pyimof.display.quiver(u, v, c=norm, bg=I0, ax=ax2,
                      cmap='jet', bg_cmap='gray')
ax2.set_title("Arrows colored according to their norm over Image")

pyimof.display.quiver(u, v, bg=norm, ax=ax3, color='r')
ax3.set_title("Red arrows over magnitude")

plt.show()
```

**Total running time of the script:** ( 0 minutes 1.070 seconds)

**Note:** Click *here* to download the full example code

## 3.3 TV-L1 vs iLK

Comparison the TV-L1 and iLK methods for the optical flow estimation.


Reference image
TV-L1 result
iLK result
Beanbags
Dimetrodon
Dogdance
Grove2
Grove3
Hydrangea
Minicooper
Rubberwhale
Urban2
Urban3
Venus
Walking

Out:

```
Processing the Beanbags sequence
        TV-L1 processing time: 2.598048sec
        ILK processing time: 0.454992sec
Processing the Dimetrodon sequence
        TV-L1 processing time: 1.877484sec
        ILK processing time: 0.313538sec
Processing the Dogdance sequence
        TV-L1 processing time: 2.664790sec
        ILK processing time: 0.458793sec
Processing the Grove2 sequence
        TV-L1 processing time: 2.905357sec
        ILK processing time: 0.456460sec
Processing the Grove3 sequence
        TV-L1 processing time: 2.968026sec
        ILK processing time: 0.458371sec
Processing the Hydrangea sequence
        TV-L1 processing time: 2.124195sec
        ILK processing time: 0.313094sec
Processing the Minicooper sequence
        TV-L1 processing time: 2.884015sec
        ILK processing time: 0.457555sec
Processing the Rubberwhale sequence
        TV-L1 processing time: 2.235224sec
        ILK processing time: 0.312935sec
Processing the Urban2 sequence
        TV-L1 processing time: 2.663834sec
        ILK processing time: 0.457162sec
Processing the Urban3 sequence
        TV-L1 processing time: 2.621930sec
        ILK processing time: 0.458131sec
Processing the Venus sequence
        TV-L1 processing time: 1.429141sec
        ILK processing time: 0.228746sec
Processing the Walking sequence
        TV-L1 processing time: 2.456856sec
        ILK processing time: 0.458667sec
```

```
from time import time
import matplotlib.pyplot as plt
import pyimof


seq_list = pyimof.data.__all__
seq_count = len(seq_list)

fig = plt.figure(figsize=((9, 2*seq_count)))
ax_array = fig.subplots(seq_count, 3)
ax_array[0, 0].set_title("Reference image")
ax_array[0, 1].set_title("TV-L1 result")
ax_array[0, 2].set_title("iLK result")
```

(continues on next page)

(continued from previous page)

```
# --- Loop over available sequences

for name, (ax0, ax1, ax2) in zip(seq_list, ax_array):

    title = name.capitalize()

    print(f"Processing the {title} sequence")

    # --- Load data
    I0, I1 = pyimof.data.__dict__[name]()

    ax0.imshow(I0, cmap='gray')
    ax0.set_ylabel(title)
    ax0.set_xticks([])
    ax0.set_yticks([])

    # --- Run TV-L1

    t0 = time()
    u, v = pyimof.solvers.tvl1(I0, I1)
    t1 = time()

    pyimof.display.plot(u, v, ax=ax1, colorwheel=False)

    print("\tTV-L1 processing time: {:02f}sec".format(t1-t0))

    # --- Run iLK

    t0 = time()
    u, v = pyimof.solvers.ilk(I0, I1)
    t1 = time()

    pyimof.display.plot(u, v, ax=ax2, colorwheel=False)

    print("\tILK processing time: {:02f}sec".format(t1-t0))

fig.tight_layout()

plt.show()
```

**Total running time of the script:** ( 0 minutes 35.773 seconds)

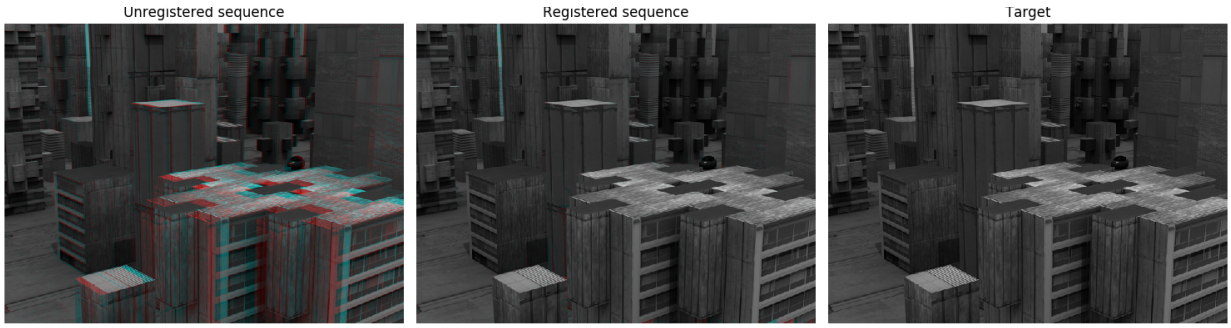
**Note:** Click *here* to download the full example code

## 3.4 Image registration

Demonstration of image registration using optical flow.

By definition, the optical flow is the vector field *(u, v)* verifying *I1(x+u, y+v) = I0(x, y)*. It can then be used for registeration by image warping.

To display registration results, an RGB image is constructed by assining the result of the registration to the red channel and the target image to the green and blue channels. A perfect registration results in a gray level image while misregistred pixels appear colored in the constructed RGB image.




```
import numpy as np
import matplotlib.pyplot as plt
from skimage.transform import warp

import pyimof

# --- Loaf the Urban2 sequence

I0, I1 = pyimof.data.urban2()

# --- Estimate the optical flow

u, v = pyimof.solvers.tvl1(I0, I1)

# --- Use the estimated optical flow for registeration

nl, nc = I0.shape

y, x = np.meshgrid(np.arange(nl), np.arange(nc), indexing='ij')

wI1 = warp(I1, np.array([y+v, x+u]), mode='nearest')

# build an RGB image with the unregistered sequence
seq_im = np.zeros((nl, nc, 3))
seq_im[..., 0] = I1
seq_im[..., 1] = I0
seq_im[..., 2] = I0

# build an RGB image with the registered sequence
reg_im = np.zeros((nl, nc, 3))
reg_im[..., 0] = wI1
reg_im[..., 1] = I0
reg_im[..., 2] = I0

# build an RGB image with the registered sequence
target_im = np.zeros((nl, nc, 3))
target_im[..., 0] = I0
target_im[..., 1] = I0
target_im[..., 2] = I0

# --- Show the result

fig = plt.figure(figsize=(15, 4))
ax0, ax1, ax2 = fig.subplots(1, 3, True)

ax0.imshow(seq_im)
```

(continues on next page)

(continued from previous page)

```
ax0.set_title("Unregistered sequence")
ax0.set_axis_off()

ax1.imshow(reg_im)
ax1.set_title("Registered sequence")
ax1.set_axis_off()

ax2.imshow(target_im)
ax2.set_title("Target")
ax2.set_axis_off()

fig.tight_layout()
plt.show()
```

**Total running time of the script:** ( 0 minutes 5.138 seconds)

CHAPTER

FOUR

# INTRODUCTION

Pyimof (for **Py**thon **im**age **o**ptical **f**low) is a pure python package for dense optical flow estimation. A good introduction to optical flow techniques can be found here.

## 4.1 Quick Example

Using Pyimov is as easy as

```
>>> from matplotlib import pyplot as plt
>>> import pyimof
>>> I0, I1 = pyimof.data.hydrangea()
>>> u, v = pyimof.solvers.tvl1(I0, I1)
>>> pyimof.display.plot(u, v)
>>> plt.show()
```

to obtain


5
10

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

# INDEX