from astropy.io import fits

import numpy as np

from scipy import signal

# read in the FITS images

image1, hdr1 = fits.getdata('image1.fits', header=True)

image2, hdr2 = fits.getdata('image2.fits', header=True)

# compute the optical flow using the Lucas-Kanade algorithm

# here we use a simple gradient filter to compute image gradients

kernel = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])

Ix = signal.convolve2d(image1, kernel, mode='same')

Iy = signal.convolve2d(image1, kernel.T, mode='same')

It = image2 - image1

# initialize the flow vectors to zero

flow\_x = np.zeros\_like(image1)

flow\_y = np.zeros\_like(image1)

# loop through each pixel and compute the velocity vector

for i in range(1, image1.shape[0]-1):

for j in range(1, image1.shape[1]-1):

A = np.array([[Ix[i, j], Iy[i, j]]])

b = -1 \* np.array([[It[i, j]]])

v = np.linalg.lstsq(A, b, rcond=None)[0]

flow\_x[i, j] = v[0]

flow\_y[i, j] = v[1]

# do something with the flow vectors, such as computing statistics or plotting

To improve the algorithm, you can try using a different optical flow algorithm or adjusting the parameters of the current algorithm, such as the window size or the maximum number of iterations. Another approach you could consider is using a different method to track the blobs, such as template matching or feature detection. It's also possible that the movement of the blobs is simply too chaotic or unpredictable to track accurately using any algorithm.

#implemented using opencv2

import numpy as np

from astropy.io import fits

import cv2

# read in the FITS file

hdulist = fits.open('data.fits')

data = hdulist[0].data

# convert the data to 8-bit grayscale

gray\_data = []

for i in range(len(data)):

gray\_data.append(cv2.normalize(data[i], None, 0, 255, cv2.NORM\_MINMAX, cv2.CV\_8U))

# initialize variables for optical flow

prev\_gray = None

prev\_pts = []

# loop through each frame of the sequence

for i in range(1, len(gray\_data)):

# detect blobs in the current frame

keypoints = cv2.SimpleBlobDetector.detect(gray\_data[i])

# get the current points and compute the optical flow

if prev\_gray is not None and len(prev\_pts) > 0:

curr\_pts, status, errors = cv2.calcOpticalFlowPyrLK(prev\_gray, gray\_data[i], prev\_pts, None)

# compute the speed of the flow vectors

flow = curr\_pts - prev\_pts

speed = np.linalg.norm(flow, axis=1)

# compute the histogram of the speeds

hist, bins = np.histogram(speed, bins=10, range=(0, np.max(speed)))

avg\_speed = np.sum(hist \* bins[:-1]) / np.sum(hist)

print('Average speed for frame', i, ':', avg\_speed)

# save the current variables for the next iteration

prev\_gray = gray\_data[i]

prev\_pts = np.array([kp.pt for kp in keypoints], dtype=np.float32)

from astropy.io import fits

import cv2

import numpy as np

# open the FITS image and get the data array

with fits.open('image.fits') as hdul:

data = hdul[0].data

# convert the data array from float64 to uint8

data\_uint8 = (data / np.max(data) \* 255).astype(np.uint8)

# initialize the background subtractor object

backSub = cv2.createBackgroundSubtractorMOG2()

while True:

# apply the background subtraction

fgMask = backSub.apply(data\_uint8)

# apply a threshold to identify bright regions

bg = cv2.cvtColor(backSub.getBackgroundImage(), cv2.COLOR\_GRAY2BGR)

fgMask = cv2.threshold(fgMask, 12, 255, cv2.THRESH\_BINARY)[1]

# remove small regions and smooth out the boundary of the remaining regions

kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (5, 5))

fgMask = cv2.morphologyEx(fgMask, cv2.MORPH\_OPEN, kernel)

fgMask = cv2.morphologyEx(fgMask, cv2.MORPH\_CLOSE, kernel)

# apply motion detection to identify regions that have moved a significant amount

contours, hierarchy = cv2.findContours(fgMask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

for cnt in contours:

(x, y, w, h) = cv2.boundingRect(cnt)

if cv2.contourArea(cnt) > 50 and cv2.mean(bg[y:y+h, x:x+w])[0] \* 5 < cv2.mean(data\_uint8[y:y+h, x:x+w])[0]:

# draw a bounding box around the region of interest

cv2.rectangle(bg, (x, y), (x + w, y + h), (0, 255, 0), 2)

# display the resulting image

cv2.imshow('image', bg)

# check for key presses

if cv2.waitKey(1) & 0xFF == ord('q'):

break

# close

import pyflow

import numpy as np

import cv2

from astropy.io import fits

# load the FITS images and convert to numpy arrays

with fits.open('image1.fits') as hdul:

frame1 = hdul[0].data

with fits.open('image2.fits') as hdul:

frame2 = hdul[0].data

frame1 = (frame1 / np.max(frame1) \* 255).astype(np.uint8)

frame2 = (frame2 / np.max(frame2) \* 255).astype(np.uint8)

# calculate the optical flow using the Farneback method

flow = pyflow.farneback(frame1, frame2, alpha=0.5, levels=3, scale=1.5,

poly\_n=7, poly\_sigma=1.1, flow\_smooth=0.5)

# visualize the flow field

mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])

hsv = np.zeros\_like(frame1)

hsv[..., 0] = ang \* 180 / np.pi / 2

hsv[..., 1] = 255

hsv[..., 2] = cv2.normalize(mag, None, 0, 255, cv2.NORM\_MINMAX)

rgb = cv2.cvtColor(hsv, cv2.COLOR\_HSV2BGR)

cv2.imshow('flow field', rgb)

cv2.waitKey(0)

Dense optical flow can be used for gaseous structures to track the motion of the gas over time and identify features such as vortices, shock waves, and turbulence. Since gaseous structures are often highly dynamic and complex, traditional tracking methods that rely on feature points or sparse optical flow may not be sufficient to capture the full extent of the motion. Dense optical flow, on the other hand, provides a dense representation of the motion field at each pixel in the image, allowing for more detailed analysis of the flow characteristics. By applying dense optical flow to gaseous structures, it is possible to gain insights into the physics of the flow and improve our understanding of the mechanisms driving the motion and evolution of gas in various astrophysical and industrial contexts.

<https://github.com/scivision/pyoptflow>

<https://github.com/pathak22/pyflow>

import cv2

from astropy.io import fits

# load the FITS images and convert to numpy arrays

with fits.open('image1.fits') as hdul:

frame1 = hdul[0].data.astype('float32')

with fits.open('image2.fits') as hdul:

frame2 = hdul[0].data.astype('float32')

# normalize the images to [0, 1] range

frame1\_norm = cv2.normalize(frame1, None, alpha=0, beta=1,

norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_32F)

frame2\_norm = cv2.normalize(frame2, None, alpha=0, beta=1,

norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_32F)

# calculate dense optical flow using the Farneback method

flow = cv2.calcOpticalFlowFarneback(frame1\_norm, frame2\_norm, None,

pyr\_scale=0.5, levels=5, winsize=11,

iterations=5, poly\_n=5, poly\_sigma=1.1,

flags=cv2.OPTFLOW\_FARNEBACK\_GAUSSIAN)

# visualize the flow field

magnitude, angle = cv2.cartToPolar(flow[..., 0], flow[..., 1])

hue = angle \* (180 / np.pi / 2)

saturation = cv2.normalize(magnitude, None, 0, 255, cv2.NORM\_MINMAX)

value = np.ones\_like(hue)

flow\_image = cv2.merge([hue, saturation, value])

flow\_image = np.asarray(flow\_image, dtype=np.uint8)

flow\_image = cv2.cvtColor(flow\_image, cv2.COLOR\_HSV2BGR)

cv2.imshow('optical flow', flow\_image)

cv2.waitKey(0)

OR MAYBE <https://github.com/ericPrince/optical-flow> ??? USES THE GUNNAR-FARNEBACK ALGORITHM

1. Coarse-to-fine optical flow: This approach involves computing the flow field at a coarse scale and then refining it at multiple finer scales, taking advantage of the hierarchical structure of the flow field. This can improve accuracy and robustness by allowing the algorithm to account for larger-scale motion and better handle varying flow patterns.

import cv2

from astropy.io import fits

# load the .FITS images and convert to numpy arrays

with fits.open('image1.fits') as hdul:

frame1 = hdul[0].data.astype('float32')

with fits.open('image2.fits') as hdul:

frame2 = hdul[0].data.astype('float32')

# normalize the input images to [0, 1] range

frame1\_norm = cv2.normalize(frame1, None, alpha=0, beta=1,

norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_32F)

frame2\_norm = cv2.normalize(frame2, None, alpha=0, beta=1,

norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_32F)

# set the parameters for coarse-to-fine optical flow calculation

pyr\_scale = 0.5

levels = 5

winsize = 11

iterations = 5

poly\_n = 5

poly\_sigma = 1.1

# set the initial flow field to zero

flow = np.zeros\_like(frame1\_norm)

# perform coarse-to-fine optical flow calculation

for i in range(levels):

# resize the images and flow field to the current level

new\_shape = (int(frame1\_norm.shape[1] \* pyr\_scale \*\* i),

int(frame1\_norm.shape[0] \* pyr\_scale \*\* i))

frame1\_level = cv2.resize(frame1\_norm, new\_shape)

frame2\_level = cv2.resize(frame2\_norm, new\_shape)

flow\_level = cv2.resize(flow, new\_shape) \* pyr\_scale

# calculate optical flow using the Farneback method

flow\_level = cv2.calcOpticalFlowFarneback(frame1\_level, frame2\_level,

flow\_level, pyr\_scale,

levels - i, winsize,

iterations, poly\_n,

poly\_sigma,

cv2.OPTFLOW\_FARNEBACK\_GAUSSIAN)

# add flow field to overall flow

flow += flow\_level

# visualize the flow field

magnitude, angle = cv2.cartToPolar(flow[..., 0], flow[..., 1])

hue = angle \* (180 / np

1. Variational optical flow: This technique formulates the optical flow computation as a variational optimization problem, where the goal is to minimize a cost function that measures the difference between the input images and the computed flow field. This approach allows for more flexible modeling of the flow field and can lead to improved accuracy and robustness.

import cv2

import numpy as np

from astropy.io import fits

from pyflow import pyflow

# load the .FITS images and convert to numpy arrays

with fits.open('image1.fits') as hdul:

frame1 = hdul[0].data.astype('float32')

with fits.open('image2.fits') as hdul:

frame2 = hdul[0].data.astype('float32')

# normalize the input images to [0, 255] range

frame1\_norm = cv2.normalize(frame1, None, alpha=0, beta=255,

norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

frame2\_norm = cv2.normalize(frame2, None, alpha=0, beta=255,

norm\_type=cv2.NORM\_MINMAX, dtype=cv2.CV\_8U)

# convert images to grayscale

frame1\_gray = cv2.cvtColor(frame1\_norm, cv2.COLOR\_BGR2GRAY)

frame2\_gray = cv2.cvtColor(frame2\_norm, cv2.COLOR\_BGR2GRAY)

# set the parameters for variational optical flow calculation

alpha = 0.01

gamma = 0.001

iterations = 50

levels = 5

delta = 0

# calculate the optical flow using the PyFlow library

u, v, \_ = pyflow.coarse2fine\_flow(frame1\_gray, frame2\_gray, alpha, gamma,

iterations, levels, delta)

# visualize the flow field

flow = np.zeros((u.shape[0], u.shape[1], 2), dtype=np.float32)

flow[:,:,0] = u

flow[:,:,1] = v

magnitude, angle = cv2.cartToPolar(flow[..., 0], flow[..., 1])

hue = angle \* (180 / np.pi / 2)

normalized\_magnitude = cv2.normalize(magnitude, None, 0, 255, cv2.NORM\_MINMAX)

rgb\_representation = cv2.cvtColor(np.float32(cv2.merge([hue, normalized\_magnitude, np.ones\_like(magnitude)])), cv2.COLOR\_HSV2BGR)

1. Discrete optimization methods: These methods use graph-based optimization techniques to compute the optimal flow field that minimizes a cost function over a set of discrete candidate vectors. This allows for efficient computation of the flow field and can result in improved accuracy and robustness compared to other methods.

import numpy as np

from scipy.ndimage import shift

from scipy.optimize import brute

def diff\_shift(image1, image2, shift):

'''Calculate the difference between two images after shifting one by a certain amount'''

shifted\_image = shift(image2, shift)

return np.sum((image1 - shifted\_image) \*\* 2)

def optimize\_shift(image1, image2):

'''Use brute force optimization to find the optimal shift for aligning two images'''

search\_range = ((-50, 50), (-50, 50)) # the range of values to search over

opt\_result = brute(diff\_shift, search\_range, args=(image1, image2), full\_output=True, finish=None)

return opt\_result[0]

# example usage

image1 = fits.getdata('image1.fits')

image2 = fits.getdata('image2.fits')

shift\_amount = optimize\_shift(image1, image2)

aligned\_image2 = shift(image2, shift\_amount)

import numpy as np

import cv2

from scipy.ndimage import shift

from scipy.optimize import brute

def diff\_shift(image1, image2, shift):

'''Calculate the difference between two images after shifting one by a certain amount'''

shifted\_image = shift(image2, shift)

return np.sum((image1 - shifted\_image) \*\* 2)

def optimize\_shift(image1, image2):

'''Use brute force optimization to find the optimal shift for aligning two images'''

search\_range = ((-50, 50), (-50, 50)) # the range of values to search over

opt\_result = brute(diff\_shift, search\_range, args=(image1, image2), full\_output=True, finish=None)

return opt\_result[0]

# example usage

image1 = cv2.imread('image1.fits', 0)

image2 = cv2.imread('image2.fits', 0)

flow = cv2.calcOpticalFlowFarneback(image1, image2, None, 0.5, 3, 15, 3, 5, 1.2, 0)

shift\_amount = optimize\_shift(image1, image2)

aligned\_optical\_flow = shift(flow, shift\_amount)

APPLYING SIMULATED ANNEALING

import numpy as np

import cv2

from scipy.ndimage import shift

from scipy.optimize import basinhopping

def diff\_shift(image1, image2, shift):

'''Calculate the difference between two images after shifting one by a certain amount'''

shifted\_image = shift(image2, shift)

return np.sum((image1 - shifted\_image) \*\* 2)

def optimize\_shift(image1, image2):

'''Use simulated annealing optimization to find the optimal shift for aligning two images'''

search\_range = ((-50, 50), (-50, 50)) # the range of values to search over

opt\_result = basinhopping(diff\_shift, [0, 0], minimizer\_kwargs={'args': (image1, image2), 'bounds': search\_range})

return opt\_result.x

# example usage

image1 = cv2.imread('image1.fits', 0)

image2 = cv2.imread('image2.fits', 0)

flow = cv2.calcOpticalFlowFarneback(image1, image2, None, 0.5, 3, 15, 3, 5, 1.2, 0)

shift\_amount = optimize\_shift(image1, image2)

aligned\_optical\_flow = shift(flow, shift\_amount)

APPLYING GENETIC ALGORITHMS

import numpy as np

import cv2

from scipy.ndimage import shift

from scipy.optimize import differential\_evolution

def diff\_shift(image1, image2, shift):

'''Calculate the difference between two images after shifting one by a certain amount'''

shifted\_image = shift(image2, shift)

return np.sum((image1 - shifted\_image) \*\* 2)

def optimize\_shift(image1, image2):

'''Use genetic algorithm optimization to find the optimal shift for aligning two images'''

search\_range = ((-50, 50), (-50, 50)) # the range of values to search over

opt\_result = differential\_evolution(diff\_shift, bounds=search\_range, args=(image1, image2), popsize=20, mutation=(0.5, 1), recombination=0.7, strategy='best1bin', maxiter=50)

return opt\_result.x

# example usage

image1 = cv2.imread('image1.fits', 0)

image2 = cv2.imread('image2.fits', 0)

flow = cv2.calcOpticalFlowFarneback(image1, image2, None, 0.5, 3, 15, 3, 5, 1.2, 0)

shift\_amount = optimize\_shift(image1, image2)

aligned\_optical\_flow = shift(flow, shift\_amount)

What is a code that does particle swarm optimization

import numpy as np

import cv2

from scipy.ndimage import shift

from pyswarm import pso

def diff\_shift(image1, image2, shift):

'''Calculate the difference between two images after shifting one by a certain amount'''

shifted\_image = shift(image2, shift)

return np.sum((image1 - shifted\_image) \*\* 2)

def optimize\_shift(image1, image2):

'''Use particle swarm optimization to find the optimal shift for aligning two images'''

search\_range = ((-50, 50), (-50, 50)) # the range of values to search over

# use single-objective PSO to minimize difference

opt\_result = pso(diff\_shift, search\_range[0], search\_range[1], args=(image1, image2))

return opt\_result

# example usage

image1 = cv2.imread('image1.fits', 0)

image2 = cv2.imread('image2.fits', 0)

flow = cv2.calcOpticalFlowFarneback(image1, image2, None, 0.5, 3, 15, 3, 5, 1.2, 0)

shift\_amount = optimize\_shift(image1, image2)

aligned\_optical\_flow = shift(flow, shift\_amount.x)