Problem Set 3

```
import pandas as pd
import numpy as np
from scipy import stats
from scipy.stats import norm
from numpy.linalg import inv
import matplotlib.pyplot as plt
import os
from pathlib import Path
import re
import statistics as stat
import seaborn as sns
import random

os.chdir('e:/MIT4/statistics-Computation/pset3')
```

4.2 Flows and correlation

```
# filex_test = 'OceanFlow/1u.csv'
# datax_test = pd.read_csv(filex_test, sep=",", header=None)
#
# datax_test.head(2)
# datax_test.shape
mask = pd.read_csv('data/mask.csv', sep=",", header=None)
nx, ny = mask.shape
nt = 100


flow_arrx = np.zeros((nx, ny, nt))
flow_arry = np.zeros((nx, ny, nt))
flow_speed_arry = np.zeros((nx, ny, nt))
flow_angle_arry = np.zeros((nx, ny, nt))

# flow_arr.shape
for i in range(nt):
        xi = pd.read_csv('data/'+str(i+1)+'u.csv',sep=",", header=None)
        yi = pd.read_csv('data/'+str(i+1)+'v.csv',sep=",", header=None)
        flow_arrx[:,:,i] = xi
        flow_arry[:,:,i] = yi
        flow_speed_arry[:,:,i] = (xi**2 + yi**2)**0.5
        flow_angle_arry[:,:,i] = np.arctan(yi/xi)



avg_flowx= np.mean(flow_arrx, axis = 2)
avg_flowy= np.mean(flow_arry, axis = 2)
avg_speed = np.mean(flow_speed_arry, axis = 2)
avg_angle = np.mean(flow_angle_arry, axis = 2)


sd_flowx = np.std(flow_arrx, axis = 2)
```

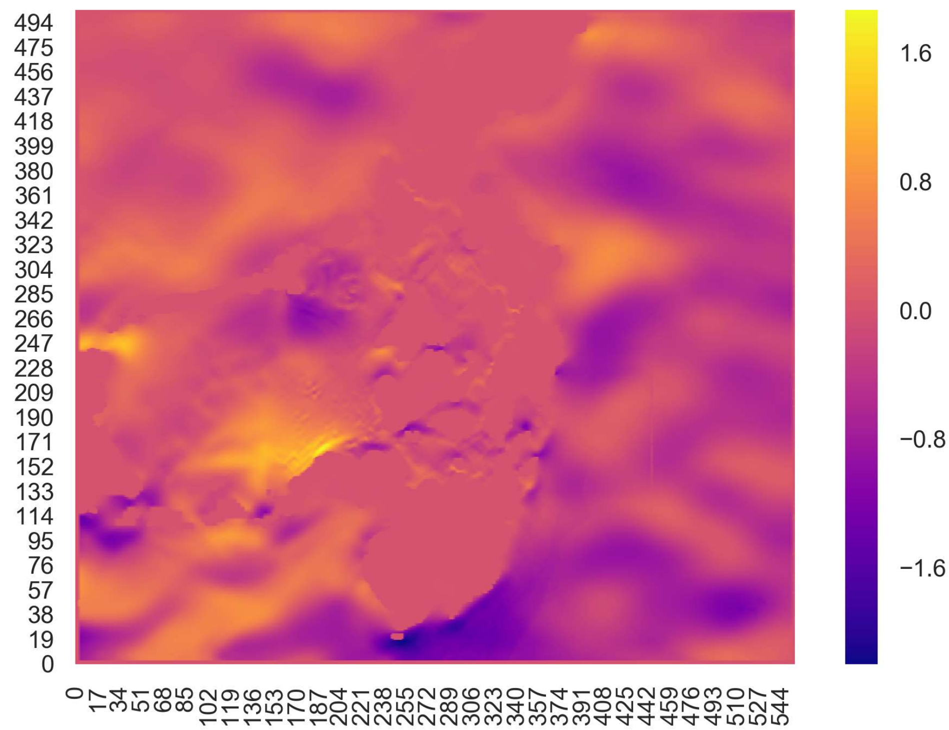
```
sd_flowy = np.std(flow_arry, axis = 2)
sd_speed = np.std(flow_speed_arry, axis = 2)
sd_angle = np.std(flow_angle_arry, axis = 2)


sns.set(font_scale=0.8)

def heatmap(mat, color, name):
        data = mat.copy()
        ax = sns.heatmap(data  ,cmap=color, square=True)
        ax.invert_yaxis()
        figure = ax.get_figure()
        figure.savefig('figure/'+str(name)+'.png', dpi=400)
```

(a)

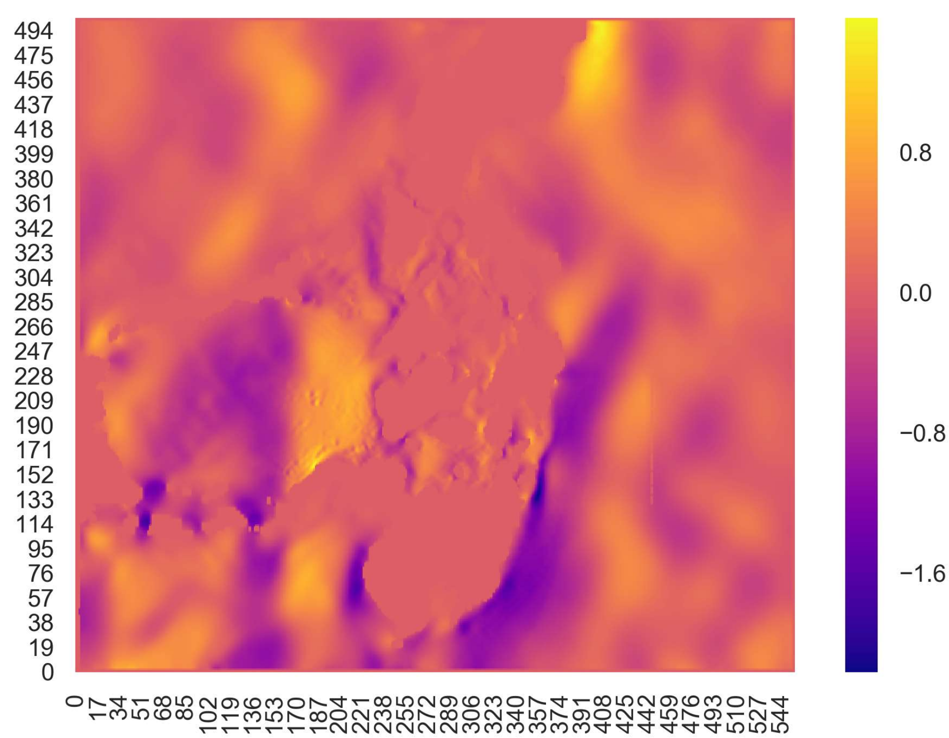
Describe the average flow (averaged over all times, and not location). Are there any constant flow currents that run in the archipelago? Compute the needed characteristics to explain your description. Again, remember, the matrix index (0; 0) will correspond in this problem to the coordinate (0km,0km), or the *bottom, left* of the plot.

```
heatmap(avg_flowx , 'plasma', 'avg_x')
heatmap(avg_flowy, 'plasma', 'avg_y')
heatmap(sd_flowx , 'plasma', 'sd_x')
heatmap(sd_flowy, 'plasma', 'sd_y')
```

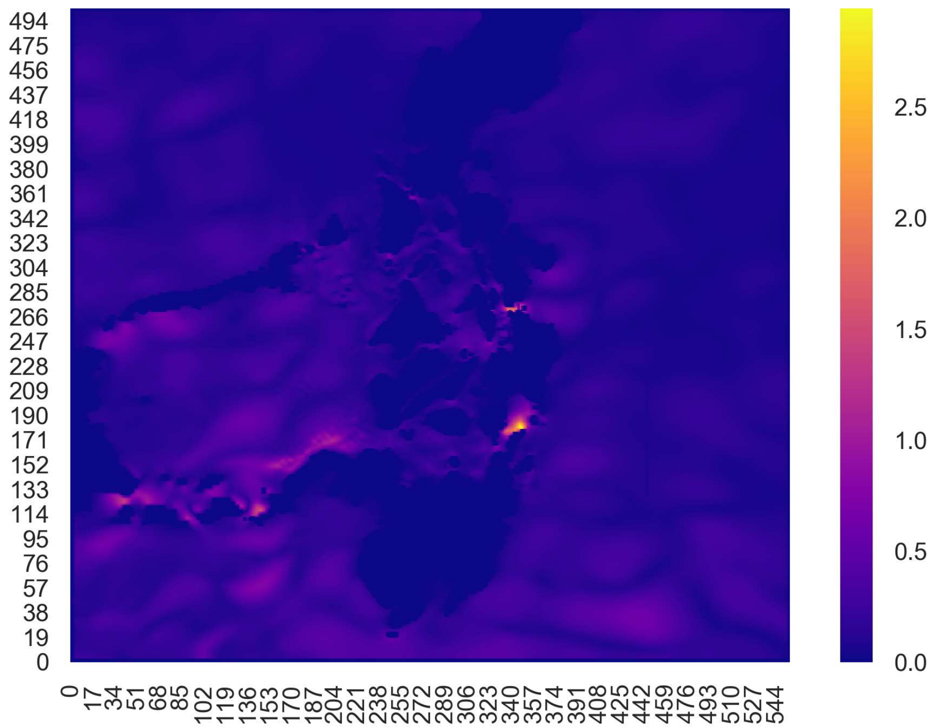
Averaged flow in u direction



Averaged flow in v direction



Standard deviation of flow speed in u direction



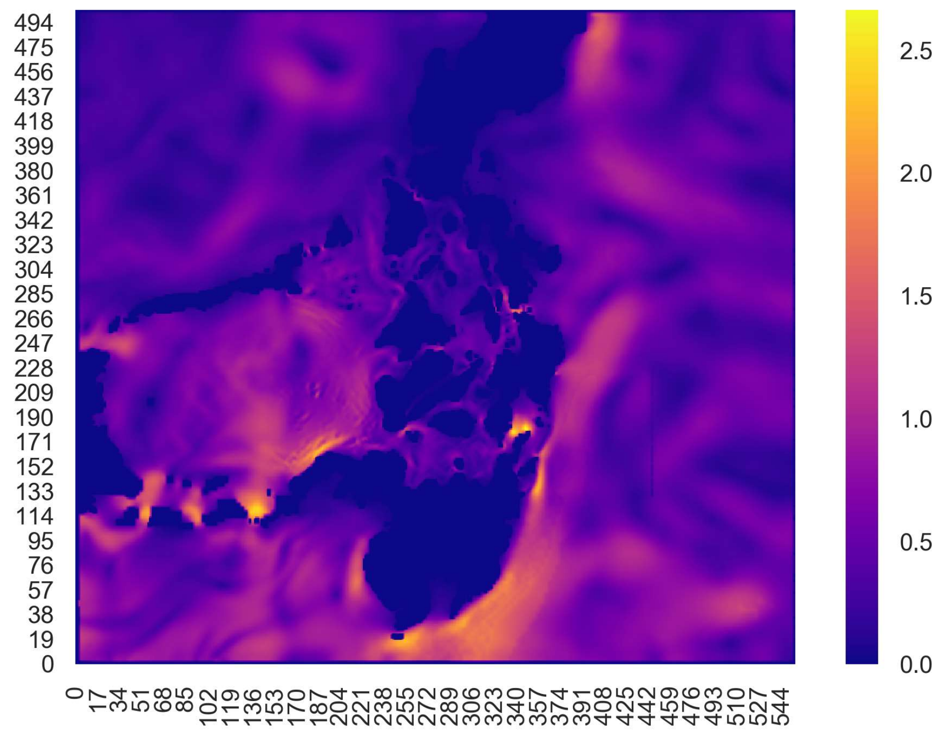
Standard deviation of flow speed in v direction ![sdv]

(b)

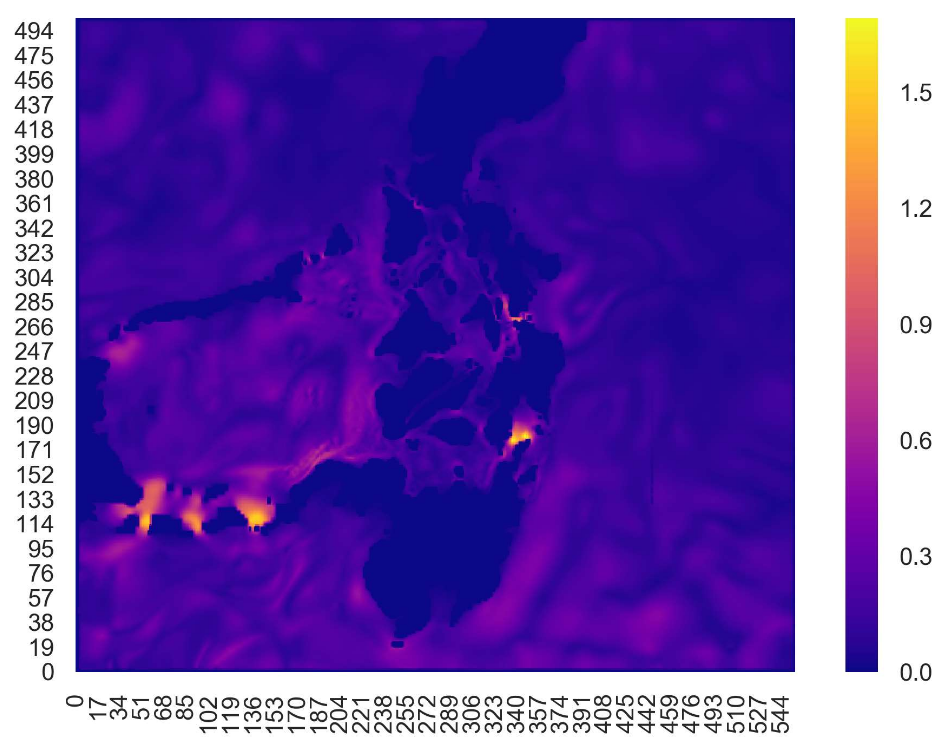
Describe the speed of the average flow (again averaged over all times). Compute the needed characteristics to explain your description.

```
heatmap(avg_speed , 'plasma', 'avg_speed')
heatmap(sd_speed , 'plasma', 'sd_speed')
heatmap(avg_angle , 'plasma', 'avg_angle')
heatmap(sd_angle , 'Purples', 'sd_angle')
```

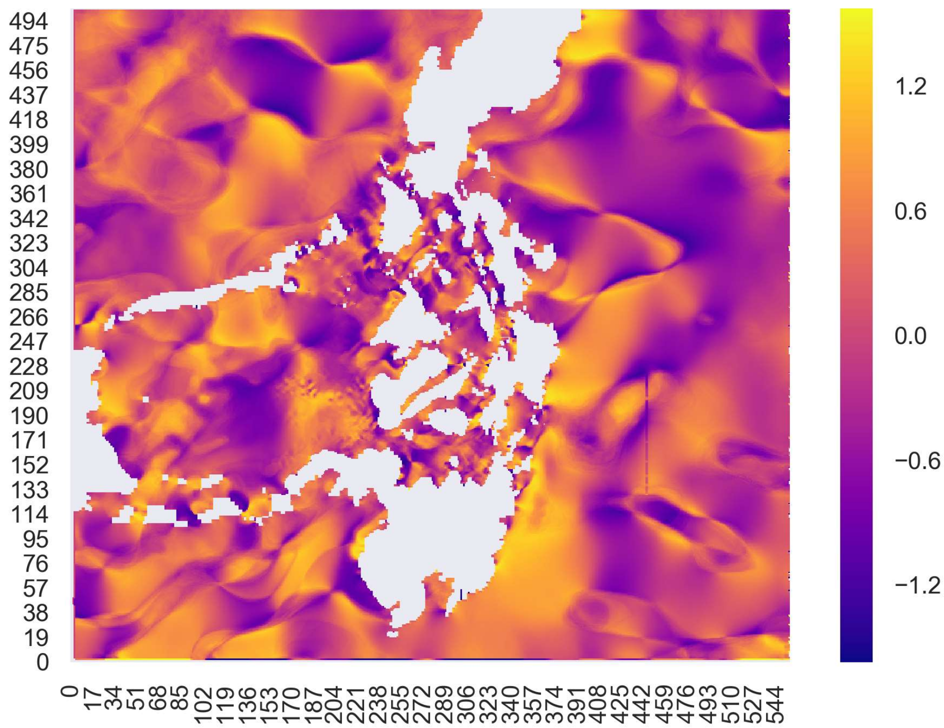
Averaged flowspeed



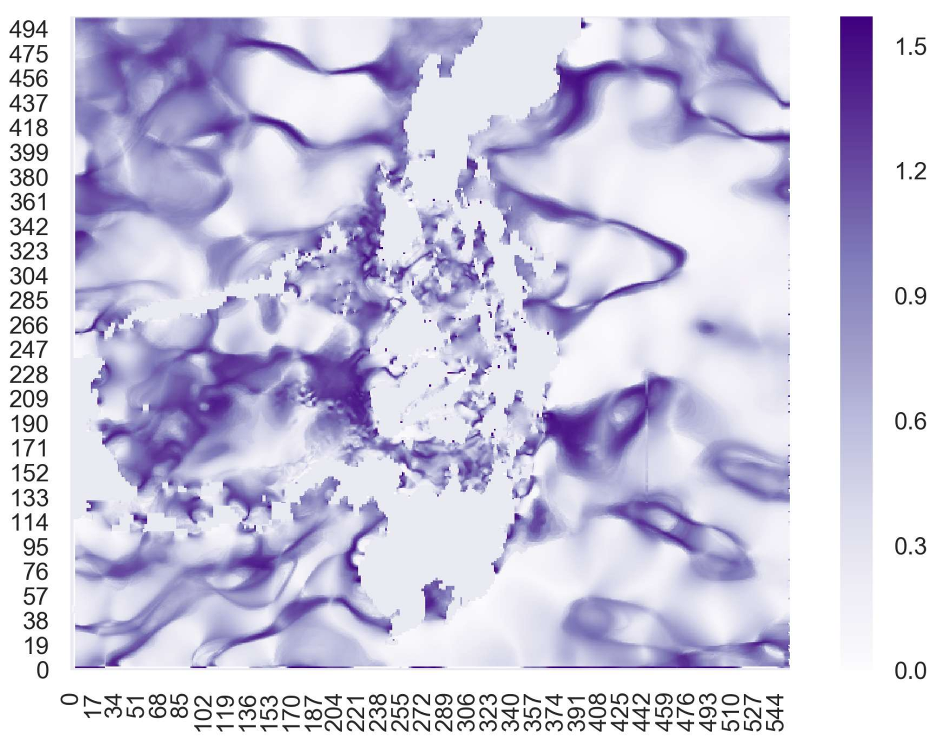
Standard deviation of flowspeed



Average flow angle (in degree)



Standard deviation of flow speed in v direction



(c)

Visualize the evolution of the flow and its speed over time. Do you observe any spatial correlation?

```
# this part only works independently or in Jupyter norebook.
%matplotlib nbagg
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

fig = plt.figure()
def f(t):
    return flow_speed_arry[:,:,t]
t = 0
im = plt.imshow(f(t), animated=True)

def updatefig(*args):
    global t
    t += 1
    im.set_array(f(t))
    return im,

ani = animation.FuncAnimation(fig, updatefig, interval=50, blit=True)
plt.show()
```

Animation animation

4.3 Predicting Trajectories

(a)

We assume that a particle in the ocean, with certain coordinates, will inherit the velocity correponding to the flow at those coordinates. Implement a procedure to track its position and movement caused by the time-varying flow. Explain the procedure, and show that it works by providing examples and plots.

```
# My procedure to track the particle position and movement is as follows:
# 1. The particle (x, y) gets its movement information from the nearest data point. To find the position of the data
# 2. We use a step-wise procedure to approximate the actual movement. In each step, we will check the position of the
# 3. To make things easier, we split the time interval (3h) to N_split steps, and we can assume that the particle wil

# Here is the sudo-code for the procedure
# (x, y, t) is a randomly chosen point in step 0 (time 0), nt is the number of snapshots (100)
# while t < nt * n_split:
#       find the coordinate (x, y)'s grid, (m, n)
#       get the speed in grid (m, n) in time t, Vx, Vy
#       update x, y, and t:
#       x += Vx * 3/n_split
#       y += Vy * 3/n_split
#       t += t + 1


def getPoint(x, y):
    m = int(round(x/3))
    n = int(round(y/3))
    return (m , n)

def getPointSpeed(x, y, t, flow_arrx, flow_arry):
    m, n = getPoint(x, y)
    Vx = flow_arrx[m, n, t]   # in  km/h
    Vy = flow_arry[m, n, t]
    return (Vx, Vy)

def simulation(x, y, flow_arrx, flow_arry, Mask = mask[::-1], nt=100, split = 30):
    '''
    simulation process given point (x,y) and nt
    '''
    t = 0
    # step = 0
    trace = []
```

```
    t_temp = 0
    nx, ny = Mask.shape
    while t < nt * split:

        trace.append([x, y])
        m, n = getPoint(x, y)
        t_n = t//split

        if (m >= nx) | (n >= ny):
            break

        if Mask.iloc[m, n]==0 :
            break

        # Vx, Vy = getSpeed(x, y, t, flow_arrx, flow_arry)
        Vy, Vx = getPointSpeed(x, y, t_n, flow_arrx, flow_arry)
        if (Vx == 0) & (Vy == 0):
            t += 1

        else:
            x = x + Vx * 3/split
            y = y + Vy * 3/split
            t += 1

    return np.array(trace)


def tracePlot(trace_list, name):
    plt.figure(figsize=(10,10))
    for trace in trace_list:
        plt.plot(trace[:,1] ,trace[:,0] , color = 'blue', marker='o',markersize = 0.06,linewidth= 0.06 )
    plt.ylim([0,1512])
    plt.xlim([0,1665])
    plt.gca().set_aspect('equal', adjustable='box')
    plt.imshow(mask[::-1],extent=[0,1665, 0,1512], origin='lower')
    plt.savefig('figure/'+str(name)+'.png', dpi=400)
    plt.show()

# trace = simulation(700, 200, flow_arrx, flow_arry, nt=100, split = 30)
# tracePlot([trace], 'testTrace')

def RandomSim( flow_arrx, flow_arry, Mask = mask[::-1], n_points = 100, n_split = 30, xmax = 1512, ymax = 1665):
    trace_list = []
    for i in range(n_points):
        x = np.random.randint(xmax)
        y = np.random.randint(ymax)

        try:
            trace = simulation(x, y, flow_arrx, flow_arry, nt=100, split = 30)
            # do it Again, just to better illustrate the pattern
            trace_dup = simulation(trace[-1][0], trace[-1][1], flow_arrx, flow_arry, nt=100, split = n_split)
            trace = np.concatenate((trace ,trace_dup))

            trace_list.append(trace )
        except:
            continue

    return trace_list

trace_list = RandomSim( flow_arrx, flow_arry, n_points = 1000, n_split = 100)
tracePlot(trace_list, 'FlowSim')
```

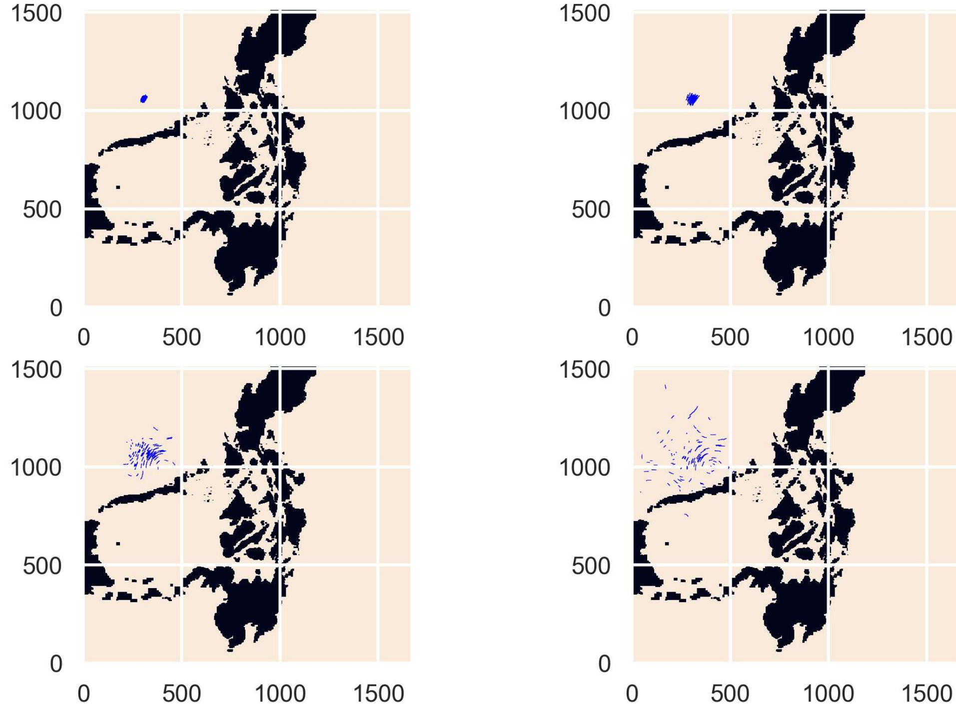
flowsim

(b)

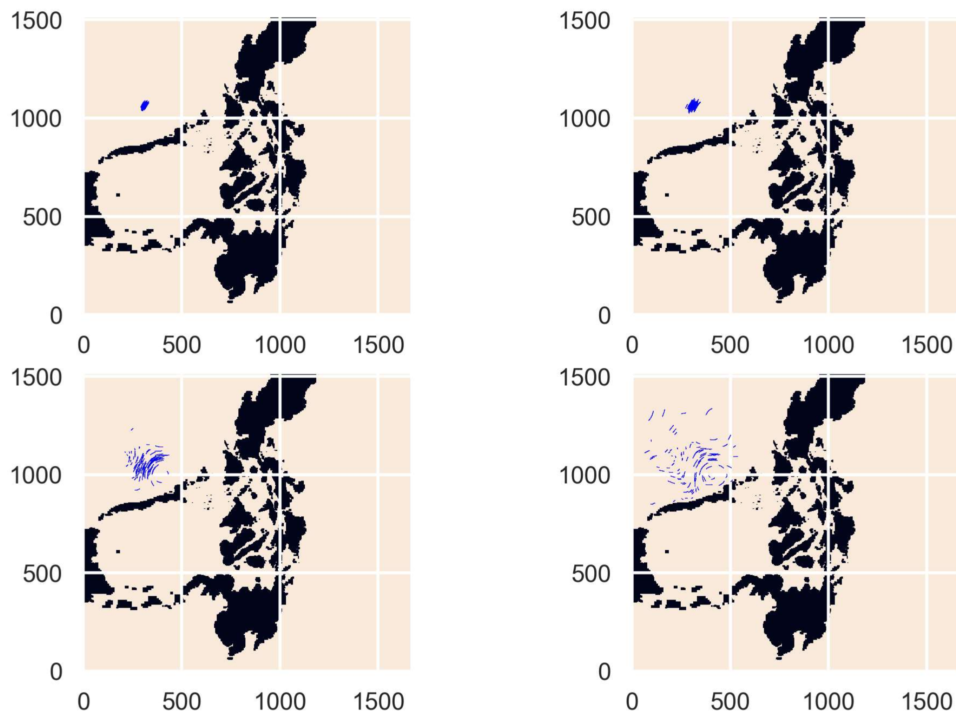
A (toy) plane has crashed in the Sulu Sea at T = 0. The exact location is unknown, but data suggests that the location of the crash follows a Gaussian distribution with mean (100; 350) (namely (300km; 1050km)) with variance sigma^2. The debris from the plane have been carried away by the ocean flow. You are about to lead a search expedition for the debris. Where would you expect the parts to be at 48hrs, 72hrs, 120hrs? Study the problem varying the variance of the Gaussian distribution. Either pick a few variance samples or sweep through the variances if desired.

```
def GaussianSim( meanx, meany, sigma, time, flow_arrx, flow_arry, Mask = mask[::-1], n_points = 100, n_split = 30, xm
    '''
    Simulation of a series of points that are Gaussian distributed with given mean and variance

    '''
    trace_list = []
    for i in range(n_points):
        x = random.gauss(meanx, sigma)
        y = random.gauss(meany, sigma)

        try:
            trace = simulation(x, y, flow_arrx, flow_arry, nt=time, split = n_split)
            # do it Again, just to better illustrate the pattern
            # trace_dup = simulation(trace[-1][0], trace[-1][1], flow_arrx, flow_arry, nt=100, split = n_split)
            # trace = np.concatenate((trace ,trace_dup))

            trace_list.append(trace )
        except:
            continue

    return trace_list

def tracePlotCompare(trace_list_l, m, n, name):
    for i, trace_list in enumerate(trace_list_l):
        plt.subplot(int(str(m)+str(n)+str(i+1)))
        for trace in trace_list:
            plt.plot(trace[:,1] ,trace[:,0] , color = 'blue', marker='o',markersize = 0.06,linewidth= 0.06 )
        plt.ylim([0,1512])
        plt.xlim([0,1665])
        plt.gca().set_aspect('equal', adjustable='box')
        plt.imshow(mask[::-1],extent=[0,1665, 0,1512], origin='lower')
    plt.savefig('figure/'+str(name)+'.png', dpi=400)
    plt.show()

# sigma = 5
trace48_5 = GaussianSim(1050, 300, 5, 16, flow_arrx, flow_arry, n_points = 100)
trace72_5 = GaussianSim(1050, 300, 5, 24, flow_arrx, flow_arry, n_points = 100)
trace120_5 = GaussianSim(1050, 300, 5, 40, flow_arrx, flow_arry, n_points = 100)

# sigma = 10
trace48_10 = GaussianSim(1050, 300, 10, 16, flow_arrx, flow_arry, n_points = 100)
trace72_10 = GaussianSim(1050, 300, 10, 24, flow_arrx, flow_arry, n_points = 100)
trace120_10 = GaussianSim(1050, 300, 10, 40, flow_arrx, flow_arry, n_points = 100)

# sigma = 50
trace48_50 = GaussianSim(1050, 300, 50, 16, flow_arrx, flow_arry, n_points = 100)
trace72_50 = GaussianSim(1050, 300, 50, 24, flow_arrx, flow_arry, n_points = 100)
trace120_50 = GaussianSim(1050, 300, 50, 40, flow_arrx, flow_arry, n_points = 100)

# sigma = 100
trace48_100 = GaussianSim(1050, 300, 100, 16, flow_arrx, flow_arry, n_points = 100)
trace72_100 = GaussianSim(1050, 300, 100, 24, flow_arrx, flow_arry, n_points = 100)
trace120_100 = GaussianSim(1050, 300, 100, 40, flow_arrx, flow_arry, n_points = 100)

tracePlotCompare([trace48_5, trace48_10, trace48_50, trace48_100], 2, 2, '48h')
tracePlotCompare([trace72_5, trace72_10, trace72_50, trace72_100], 2, 2, '72h')
tracePlotCompare([trace120_5, trace120_10, trace120_50, trace120_100], 2, 2, '120h')
```

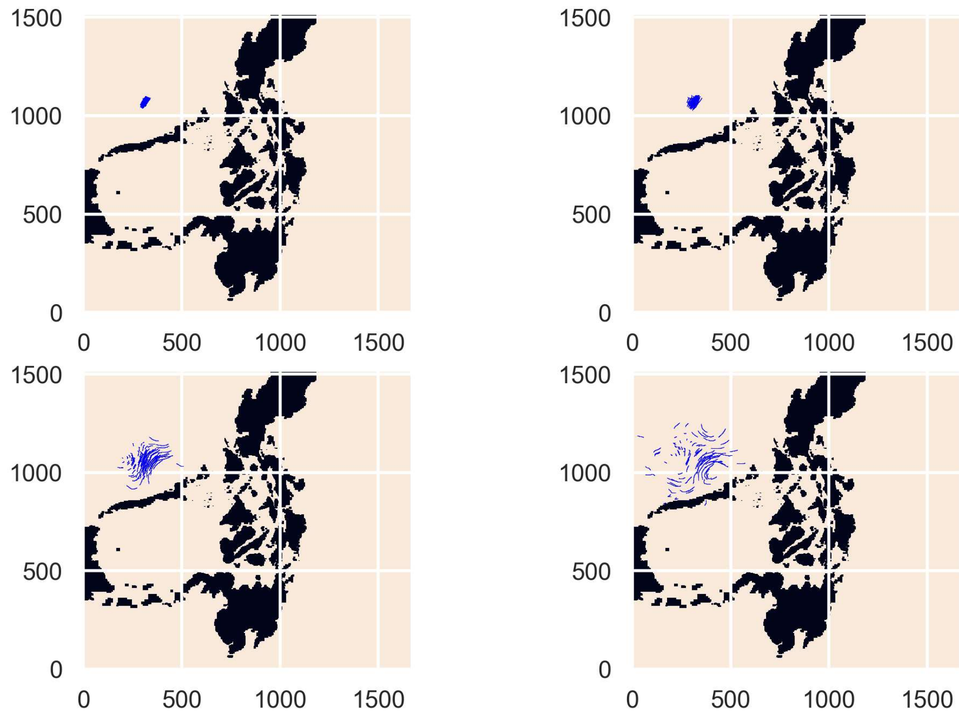
48h simulation:



72h simulation:



120h simulation:



4.4 Path Planning

(a)

Devise and implement a scheme to plan a route that minimizes travel time. The vehicle consumes 1 unit of fuel for every 1 unit of time the engine is on. Plan a route from (x0; y0) = (70; 400) to (xf ; yf) = (360; 170). How does the route vary for different values of V ? You are not required to find the optimal solution, but a very good solution.

```
import networkx as Netx

# read the data
u40 = pd.read_csv('data/40u.csv',sep=",", header=None)
v40 = pd.read_csv('data/40v.csv',sep=",", header=None)
speed40 = np.sqrt(u40 ** 2 + v40 ** 2)
mask = pd.read_csv('data/mask.csv', sep=",", header=None)


plt.imshow(u40, origin = 'lower')
plt.imshow(mask,  origin = 'lower')
Mask = mask[::-1]
plt.imshow(Mask,  origin = 'lower')

def ConstructGraph(Mask, v_boat, df_v = speed40):
    # Construct the Graph
    G = Netx.Graph()

    # Coordinate matrix
    coord_mat = np.arange(Mask.shape[0] * Mask.shape[1]).reshape(Mask.shape)

    # make each entry of the Mat a Graph node
    for i in range(Mask.shape[0]):
        for j in range(Mask.shape[1]):
            if Mask.iloc[i , j] != 0:
```

```
                code = coord_mat[i, j]
                G.add_node(code, position = (i * 3, j * 3), coordx = i,coordy= j)

    # add edges
    for node, data in G.nodes(data=True):

        v_node = df_v.iloc[int(data['coordx']), int(data['coordy'])]
        try:
            t_total = 3/(v_node + v_boat)
        except:
            t_total = np.Inf

        if (data['coordx'] < Mask.shape[0]-1) &  ((node + 1) in G.nodes()):
            G.add_edge(node, node + 1, weight = t_total) # undirected graph, add edge linking the right node

        if (data['coordy'] < Mask.shape[1]-1) & ((node + Mask.shape[0]) in G.nodes()):
            G.add_edge(node, node + Mask.shape[0], weight = t_total)

    return G


def findPath(startx, starty, endx, endy, Mask, v_boat, df_v = speed40):

    coord_mat = np.arange(Mask.shape[0] * Mask.shape[1]).reshape(Mask.shape)
    start_coord = coord_mat[getPoint(startx, starty)]
    end_coord = coord_mat[getPoint(endx, endy)]

    graph = ConstructGraph(Mask, v_boat, df_v)
    # find the shortest path
    shorstpath = Netx.dijkstra_path(graph, start_coord, end_coord, 'weight')
    shorstpath_len = Netx.dijkstra_path_length(graph, start_coord, end_coord, 'weight')
    print('Shortest Path Length with boat speed = '+str(v_boat)+ ' : '+ str(shorstpath_len))
    # return (shorstpath, shorstpath_len)



findPath(70, 400, 360, 170, Mask, v_boat=5, df_v = speed40)
findPath(70, 400, 360, 170, Mask, v_boat=10, df_v = speed40)
findPath(70, 400, 360, 170, Mask, v_boat=50, df_v = speed40)
findPath(70, 400, 360, 170, Mask, v_boat=100, df_v = speed40)
```

Shortest Path Length with boat speed = 5 : 1495.9122597210715 Shortest Path Length with boat speed = 10 : 811.439630624186 Shortest Path Length with boat speed = 50 : 174.18555795602148 Shortest Path Length with boat speed = 100 : 87.90732176442386

(b)

Describe a potential scheme that would compute the shortest path in a time varying flow. Specically, reconsider (a) while working with the whole data set. Explain the different pieces of the algorithm.

Answer:

As the flow speed in each grid changes over time, it is hard to use a static graph to find the shortest path. One possible solution is the greedy algorithem that find the nearest node in each step:

```
current_node = start_node
dist = 0
t = 0
min_dist = min_d(start_node), min_d(node) is the function that return the nearest node_i in adjacent nodes and the cc
while current_node != end_node:
    node_i, dist_i = min_d(current_node)
    current_node = node_i
    dist = dist + dist_i
    t = t + 1
```

Or we can use the average speed and construct a static graph.