Problem 2: [10 bonus pts] Identifying long-range correlations.

In this problem, we will try to identify areas in the Philippine Archipelago with long-range correlations. Your task is to identify two places on the map that are not immediately next to each other but still have some high correlation in their flows. Your response should be the map of the Archipelago with the two areas marked (e.g., circled). You claim that those two areas have correlated flows. Explain how did you found those two areas have correlated flows.

We start with the explanation on how the correlation was computed.

data_u and data_v are two three-dimensional numpy arrays, with first dimension being time and other two – row and column position of the initial dataset

```
print(data_u.shape)
print(data_v.shape)
```

```
(100, 555, 504)
(100, 555, 504)
```

We then forming a set of all grid points, randomly sample from it and compute correlation coefficients

```
[7]  import random
     random.seed(0)

     # Forming set of all grid points
     grids = []
     step = 1
     for i in range(0, data_u.shape[1], step):
       for j in range(0, data_u.shape[2], step):
         grids.append((i, j))

     corr_coeff_u = defaultdict()
     corr_coeff_v = defaultdict()

     # Randomly select the points
     for _ in range(1000000):
       x1, y1 = random.choice(grids)
       x2, y2 = random.choice(grids)
       corr_coeff_u[(x1, y1, x2, y2)] = np.corrcoef(data_u[:, x1, y1], data_u[:, x2, y2], rowvar=False)
       corr_coeff_v[(x1, y1, x2, y2)] = np.corrcoef(data_v[:, x1, y1], data_v[:, x2, y2], rowvar=False)
```

If the average correlation coefficient exceeds 0.97 and two points have at least 300km distance between them, then these two points are selected and added to the “lines” list:

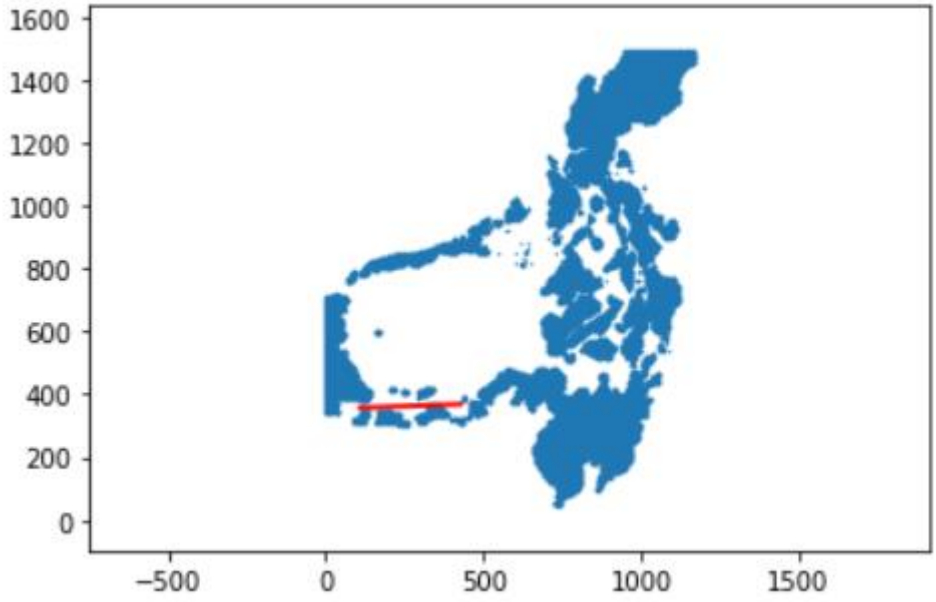
```
threshold = 0.95

lines = []
for coords in corr_coeff_u.keys():
  x1 = coords[0]
  y1 = coords[1]
  x2 = coords[2]
  y2 = coords[3]
  if ((corr_coeff_u[coords][0, 1] + corr_coeff_v[coords][0, 1])/2 >= threshold
      and np.sqrt((x1*3 - x2*3)**2 + (y1*3 - y2*3)**2) > 300):
    lines.append([(x1*3, y1*3), (x2*3, y2*3)])
```

For the above conditions, only one pair of grid points is found (values are in km):

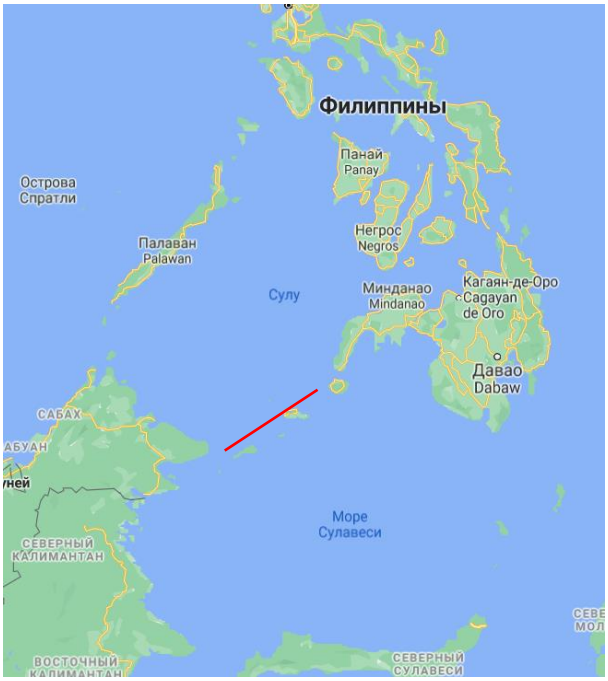
```
print(lines)
```

```
[[(102, 357), (435, 369)]]
```

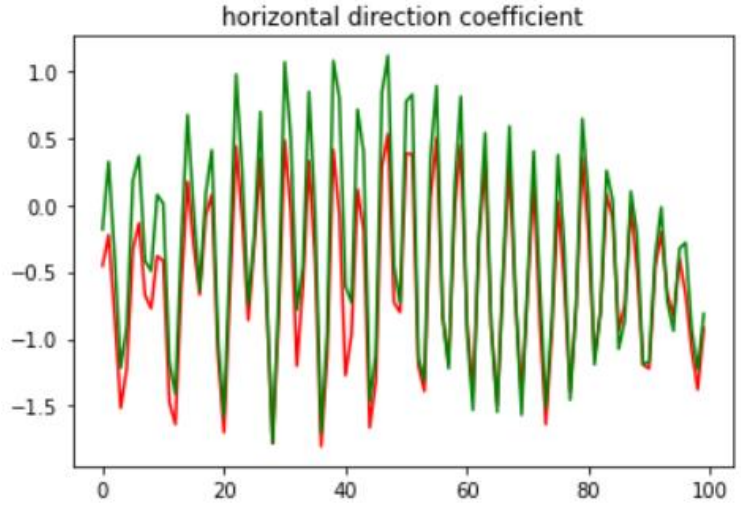
We then plot land as a scatter plot for all the points with zero variance.

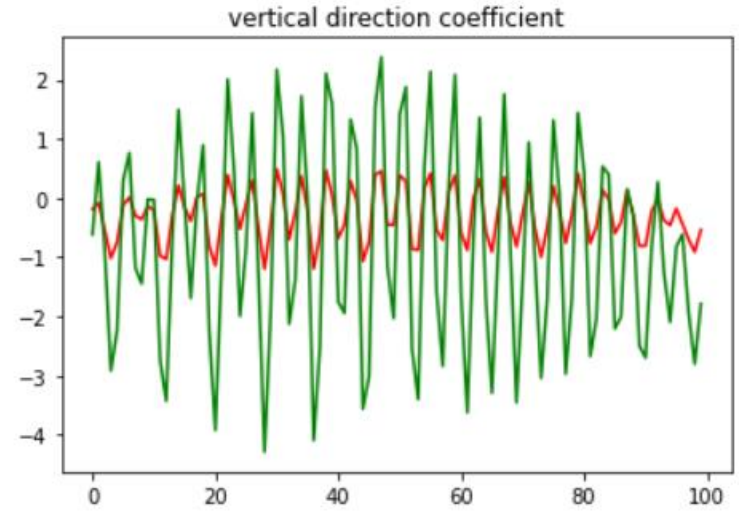


We can observe that two areas located at the chain of islands are positively correlated. The most convincing reason of why this is the case are tides, as both points are located close to the coasts where the tide-effect is the strongest.



The two correlation plots for horizontal and vertical directions below speak in favor to this hypothesis, as the oscillation period is about 24 hours.





With red curve being the flow in the more western point in the google-maps picture and green curve corresponding to the eastern point.

Problem 3: [20 pts] Simulating particle movement in flows.

In this problem, you are asked to build a simulator that can track a particle's movement on a time-varying flow.

Problem 3.a:

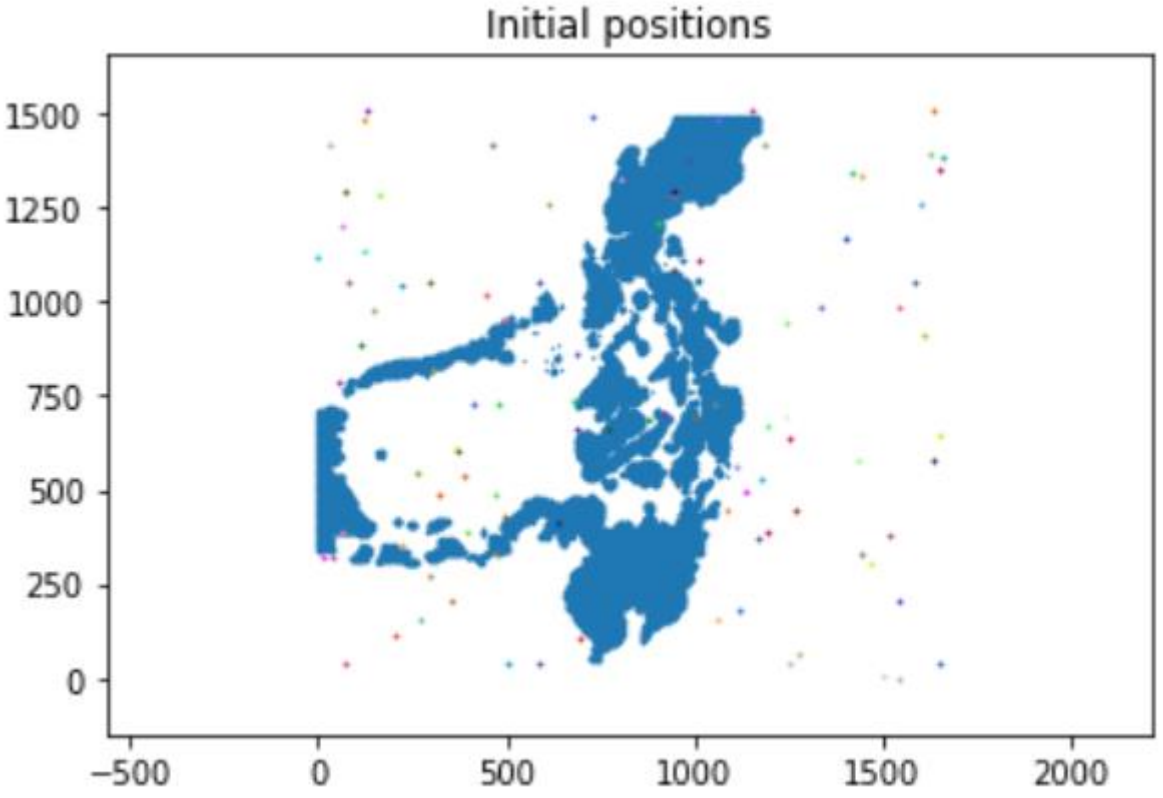
We assume that the velocity of a particle in the ocean, with certain coordinates, will be determined by the corresponding water flow velocity at those coordinates. Implement a procedure to track the position and movement of multiple particles as caused by the time-varying flow given in the data set. Explain the procedure, and show that it works by providing examples and plots.

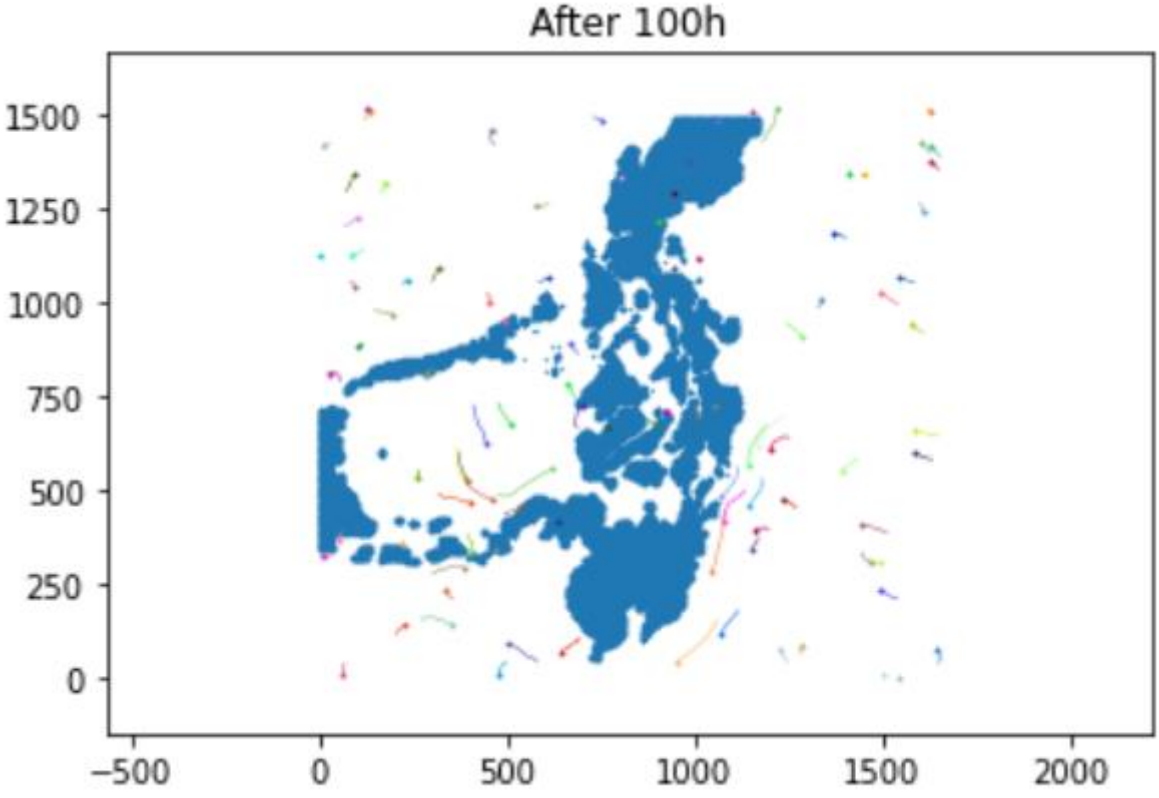
We start by forming the two main dictionaries X and Y to which the keys will be the tuples with particle initial coordinates and the values are 100-dimensional numpy arrays with coordinate for each time stamp:

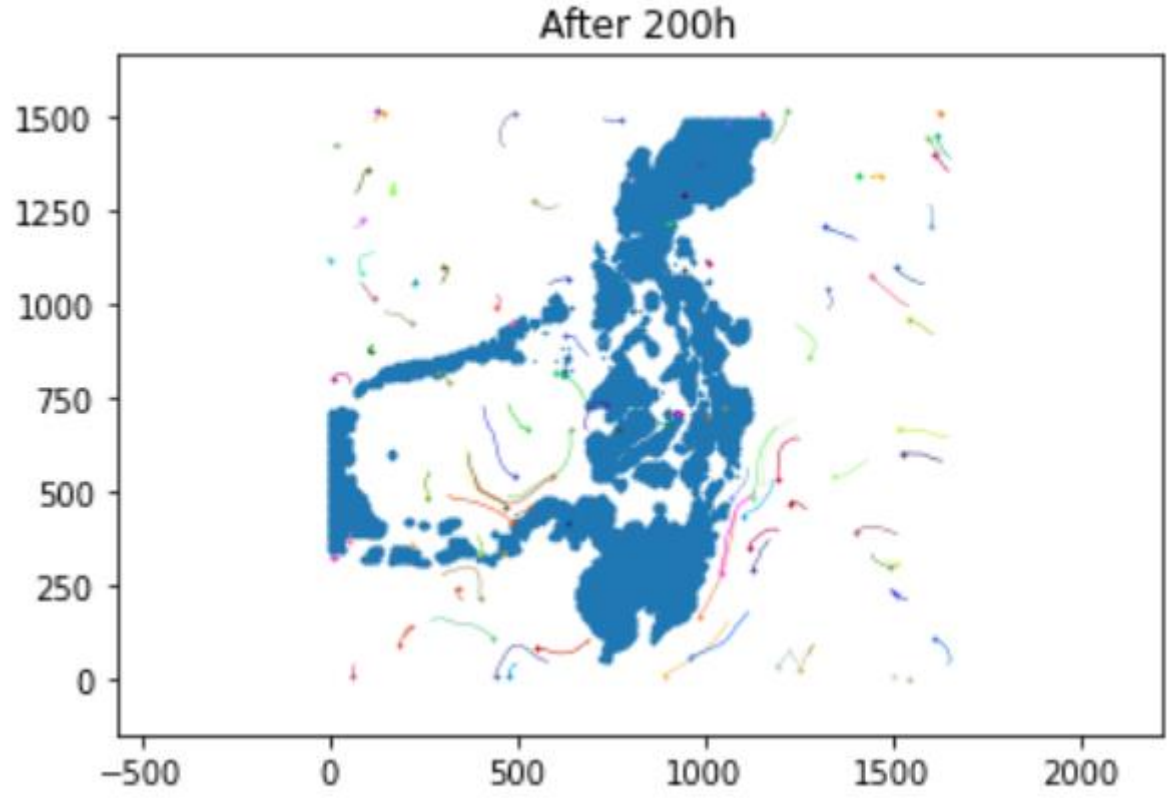
```
[128] # Building a flow simulator
      particles = random.sample(grids, 100)
      x = defaultdict()
      y = defaultdict()
      for particle in particles:
        row, column = particle
        x0 = row*3
        y0 = column*3
        time_step = 1/18

        x[particle] = np.zeros(100).reshape(100, 1)
        y[particle] = np.zeros(100).reshape(100, 1)

        for time in range(0, 100):
          x[particle][time, 0] = x0 + data_u[time, row-1, column-1]*3
          y[particle][time, 0] = y0 + data_v[time, row-1, column-1]*3
          x0 = x[particle][time, 0]
          y0 = y[particle][time, 0]
          row = int(np.round(x0/3))
          column = int(np.round(y0/3))
```

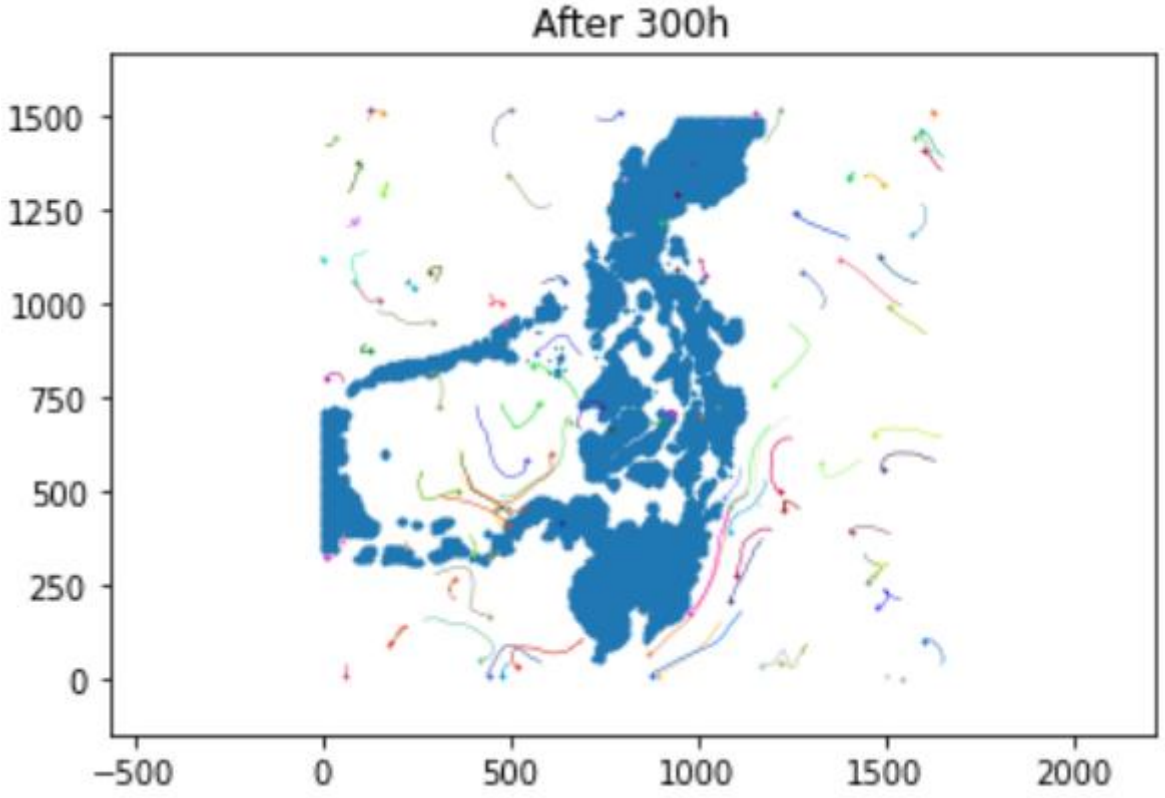
The last two columns are particularly important as it rounds the position of the next grid to the closest existing grid and then the flow on the next step is calculated accordingly

Below are the plots for initial stage and for the stages after 100h, 200h and 300h


Initial positions
1500
1250
1000
750
500
250
0
−500
0
500
1000
1500
2000

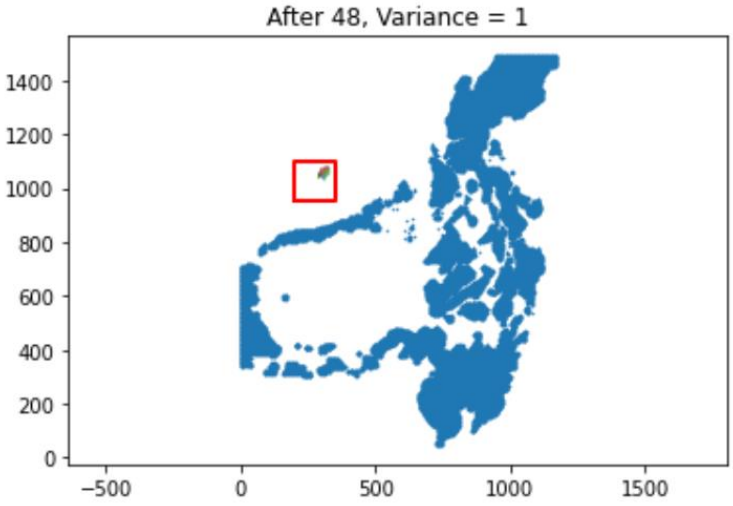

After 100h
1500
1250
1000
750
500
250
0
−500
0
500
1000
1500
2000

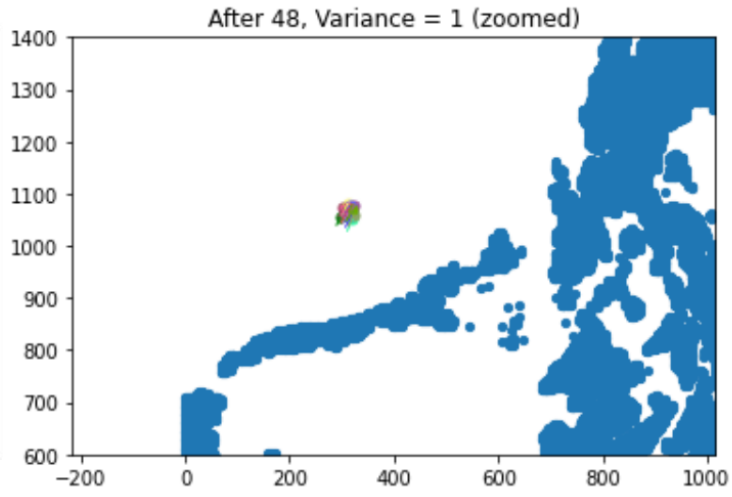

After 200h
1500
1250
1000
750
500
250
0
−500
0
500
1000
1500
2000

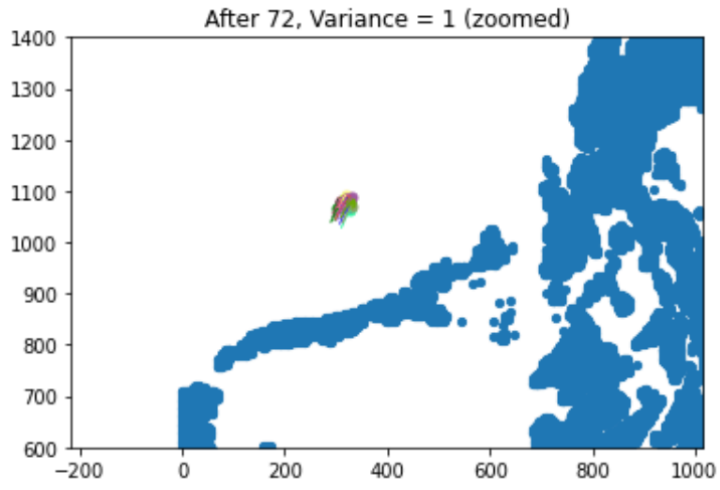

After 300h
1500
1250
1000
750
500
250
0
−500
0
500
1000
1500
2000

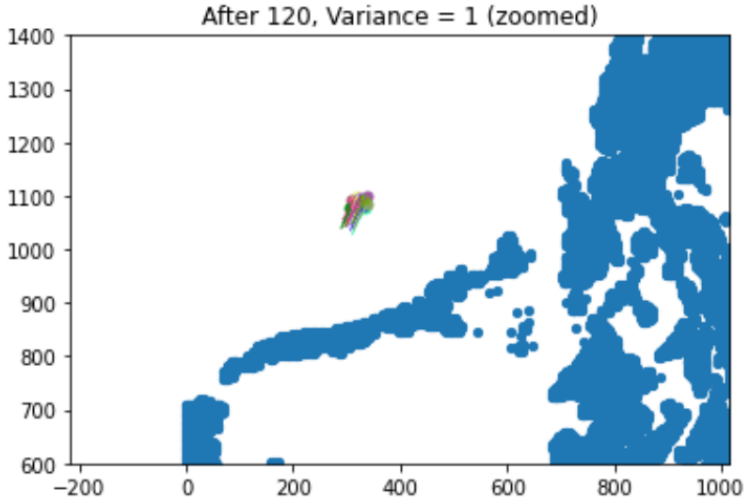
Problem 3.b:[10 points]

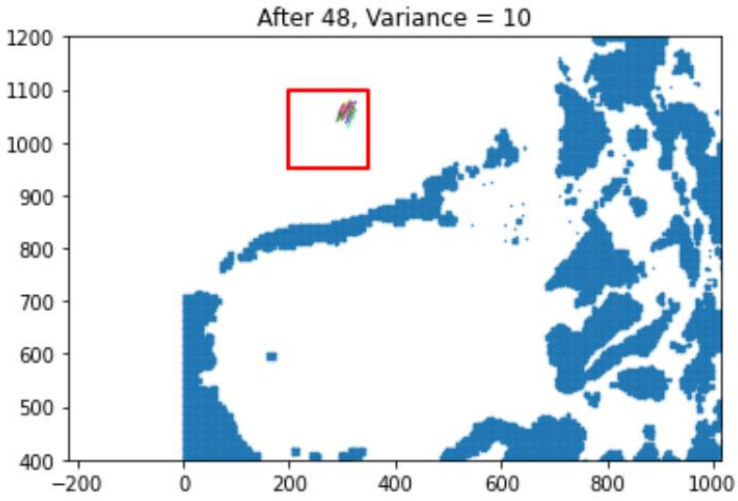
A (toy) plane has crashed in the Sulu Sea at T=0. The exact location is unknown, but data suggests that the location of the crash follows a Gaussian distribution with
mean (100,350) (namely (300km,1050km)) with variance σ2. The debris from the plane has been carried away by the ocean flow. You are about to lead a search expedition for the debris. Where would you expect the parts to be at 48hrs, 72hrs, 120hrs? Study the problem by varying the variance of the Gaussian distribution. Either pick a few variance samples or sweep through the variances if desired. (*Hint:* Sample particles and track their evolution.)

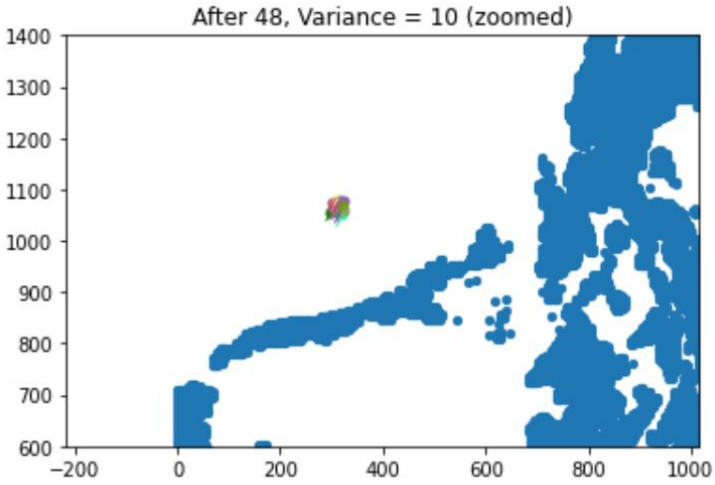


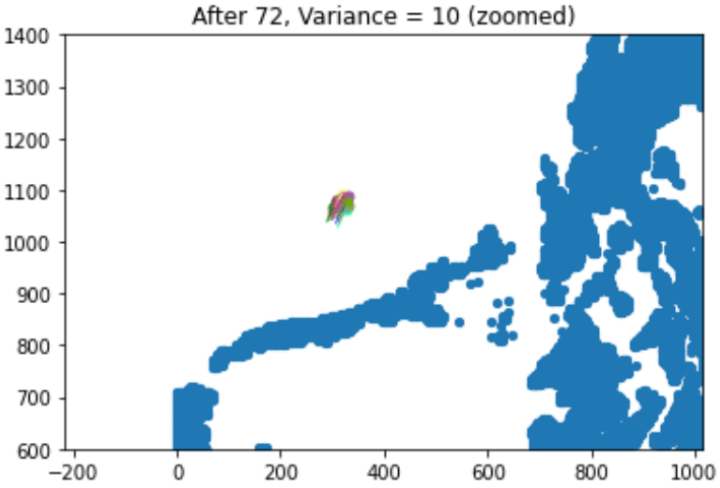


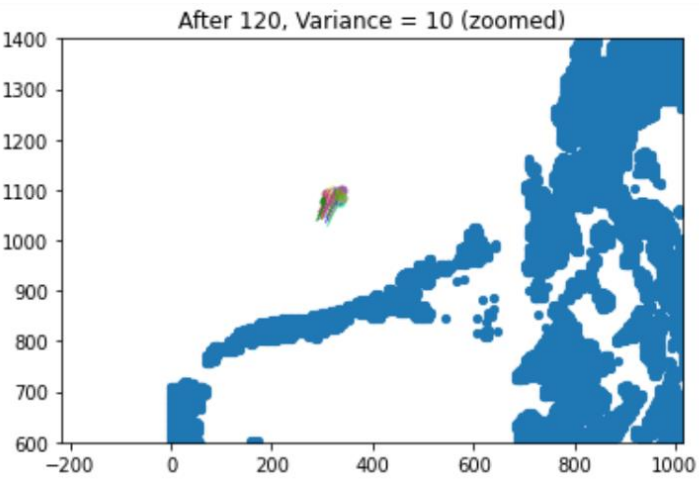


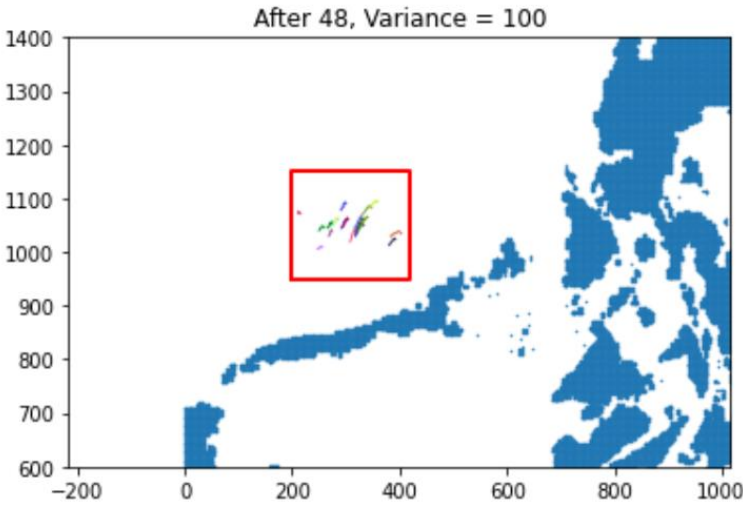


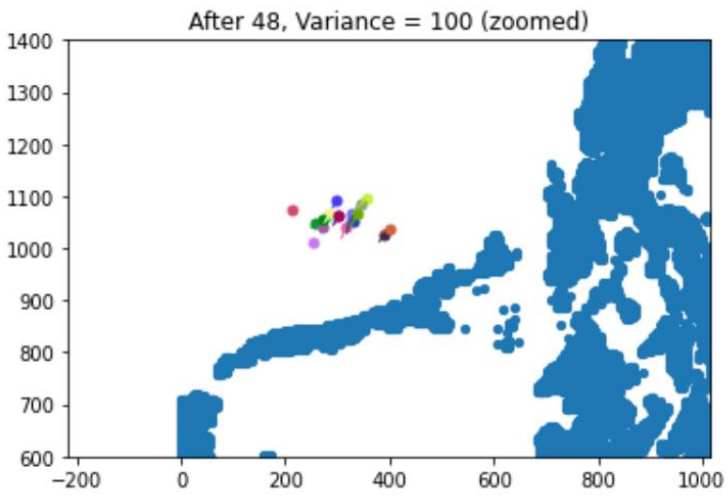

After 48, Variance = 10
1200
1100
1000
900
800
700
600
500
400
−200
0
200
400
600
800
1000

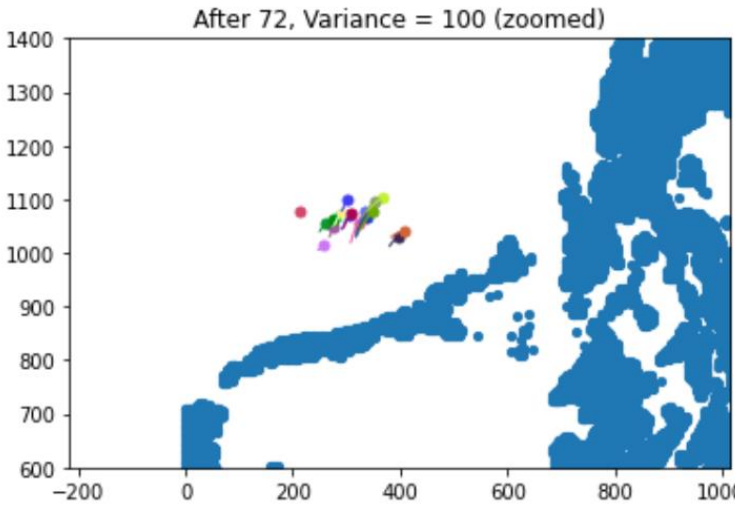

After 48, Variance = 10 (zoomed)
1400
1300
1200
1100
1000
900
800
700
600
−200
0
200
400
600
800
1000

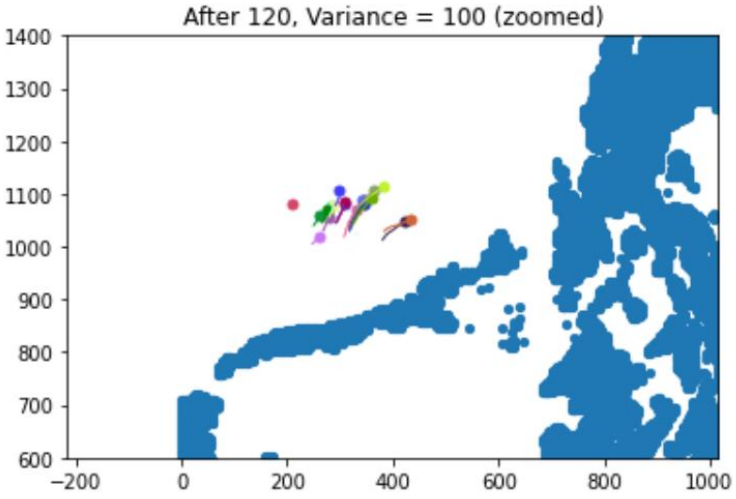

After 72, Variance = 10 (zoomed)
1400
1300
1200
1100
1000
900
800
700
600
−200
0
200
400
600
800
1000


After 120, Variance = 10 (zoomed)
1400
1300
1200
1100
1000
900
800
700
600
−200
0
200
400
600
800
1000









If the location of the plane crash is determined with high confidence, then the research region is not more than 50kmx50km, and shifted from the direction of initial crash (300,1150) on 50km to the nord-nord-west.

If in turn, the variance is quite high (say 100 as in our example), then the research region is much higher: from 1000km to 1100km in altitude and from 220km to 420km in longitude, that is 8 times larger. If the plane location is not certain, the general trend for the debris flow in this region is also nord-nord-west, with dominance of western direction for the most western possible crash points.

I didn't succeed to overcome the singularities in covariance matrix to answer questions 4a, 4b and 4c, I am therefore continuing using the sklearn gp library

Problem 4.d

Currently, most of the commonly used languages like Python, R, Matlab, etc., have pre-installed libraries for Gaussian processes. Use one library of your choice, maybe the language or environment you like the most, and compare the obtained results. Did you get the same parameters as in problem 4.a? If not, why are they different? Elaborate on your answer.

The scikit-learn' library Gaussian_process has been used

```
[ ]  # Gaussian Process using sklearn libriary
     import sklearn.gaussian_process as gp
     kernel_u = gp.kernels.ConstantKernel(1.0, (1e-1, 1e3)) * gp.kernels.RBF(10.0, (1e-3, 1e3))
     kernel_v = gp.kernels.ConstantKernel(1.0, (1e-1, 1e3)) * gp.kernels.RBF(10.0, (1e-3, 1e3))
     tau = 0.001
     model_u = gp.GaussianProcessRegressor(kernel=kernel_u, n_restarts_optimizer=10, alpha=tau, normalize_y=True)
     model_v = gp.GaussianProcessRegressor(kernel=kernel_v, n_restarts_optimizer=10, alpha=tau, normalize_y=True)
     model_u.fit(X_train, y_train_u)
     model_v.fit(X_train, y_train_v)
     params_u = model_u.kernel_.get_params()
     params_v = model_v.kernel_.get_params()
     y_pred_all_u, std = model_u.predict(np.arange(100).reshape(-1, 1), return_std=True)
     y_pred_all_v, std = model_v.predict(np.arange(100).reshape(-1, 1), return_std=True)
     y_pred_test_u, std = model_u.predict(X_test, return_std=True)
     y_pred_test_v, std = model_v.predict(X_test, return_std=True)
     #y_pred_test, cov = model.predict(X_test, return_cov=True)
     MSE_u = ((y_pred_test_u-y_test_u)**2).mean()
     MSE_v = ((y_pred_test_v-y_test_v)**2).mean()
     print("tau=", tau, "     Mean squared error U-direction:", MSE_u)
     print("tau=", tau, "     Mean squared error V-direction:", MSE_v)
     print("Variance (constant term) U-direction", params_u['k1'])
     print("Variance (constant term) V-direction", params_v['k1'])
     print("Bandwidth (RBF term) U-direction", params_u['k2'])
     print("Bandwidth (RBF term) V-direction", params_v['k2'])
```

```
tau= 0.001      Mean squared error U-direction: 0.0012474431128541735
tau= 0.001      Mean squared error V-direction: 0.00037707275293039647
Variance (constant term) U-direction 0.316**2
Variance (constant term) V-direction 0.316**2
Bandwidth (RBF term) U-direction RBF(length_scale=3.42)
Bandwidth (RBF term) V-direction RBF(length_scale=3.81)
```

The details of the above code:

Kernel consists of two kernels – constant and RBF. Each has one parameter passed into and the search space for its optimization: the constant parts has the variance parameter and the RBF part – the length scale parameter

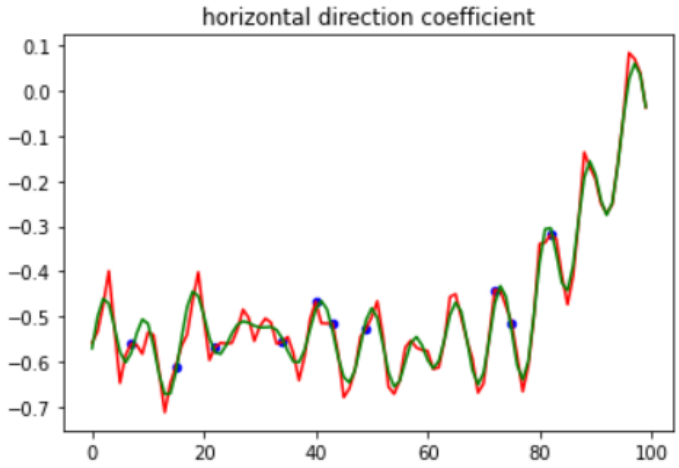
We then fit the model to the training data with .fit method

The GP library uses marginal loglikelihood as the main metrics for optimization

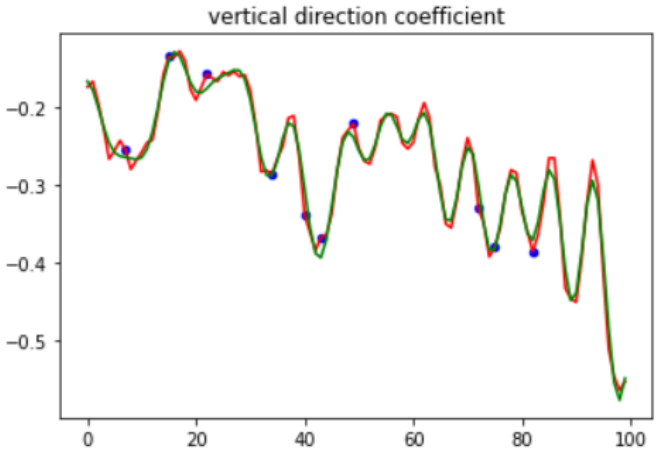
The optimal parameters are printed below the code

The goodness of fit can be estimated visually by plotting the predictions: The red curve corresponds to the real data, the green curve is the predictions, the blue points are the unseen test points

```
import matplotlib.pyplot as plt
fig, ax1 = plt.subplots(1, 1)
ax1.plot(range(100), flow_u, color = 'r')
ax1.plot(range(100), y_pred_all_u, color = 'g')
ax1.scatter(X_test, y_test_u, color = 'b', s = 20)
ax1.set_title("horizontal direction coefficient")
plt.show()
```




```
import matplotlib.pyplot as plt
fig, ax1 = plt.subplots(1, 1)
ax1.plot(range(100), flow_v, color = 'r')
ax1.plot(range(100), y_pred_all_v, color = 'g')
ax1.scatter(X_test, y_test_v, color = 'b', s = 20)
ax1.set_title("vertical direction coefficient")
plt.show()
```

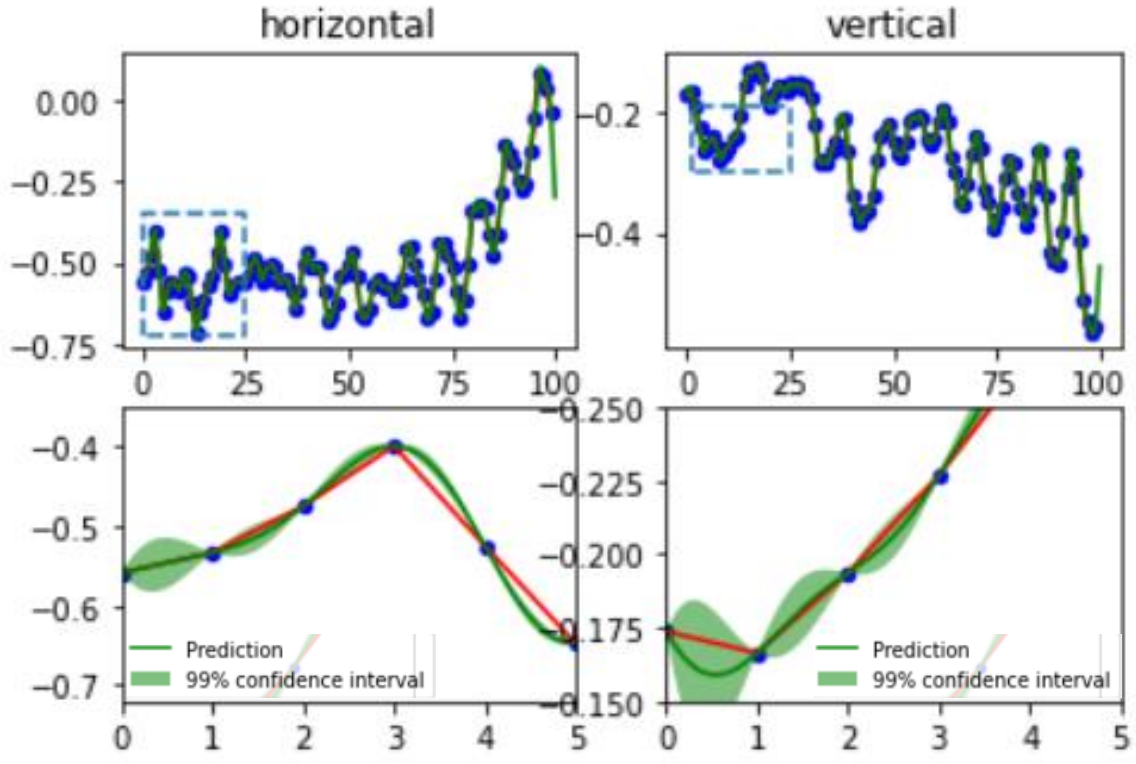


Problem 5. Estimating unobserved flow data.

We then proceed to fitting the seen data to interpolate onto the unseen data. To have a smooth confidence bounds, we choose 1000 points between each to observations. We then fit the model using seen data and predict the flow values at the unseen data points

```
tau = 0.0000001

X_unseen = np.arange(0, 100, 0.001).reshape(-1,1)
X_seen = np.arange(0, 100, 1).reshape(-1,1)

model_u = gp.GaussianProcessRegressor(kernel=kernel_u, n_restarts_optimizer=10, alpha=tau, normalize_y=True)
model_v = gp.GaussianProcessRegressor(kernel=kernel_v, n_restarts_optimizer=10, alpha=tau, normalize_y=True)

model_u.fit(X_seen, flow_u[X_seen])
model_v.fit(X_seen, flow_v[X_seen])

y_pred_unseen_u, std_u = model_u.predict(X_unseen, return_std=True)
y_pred_unseen_v, std_v = model_v.predict(X_unseen, return_std=True)
```

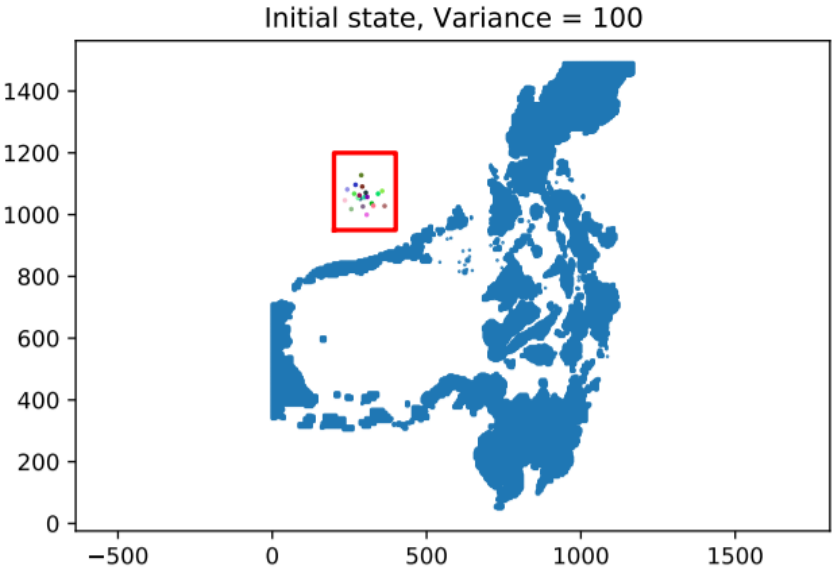
The tau has been modified to clearly see the change in the confidence interval in the observed points. For the value of 0.001 the 3σ is just the equidistant of the prediction line.

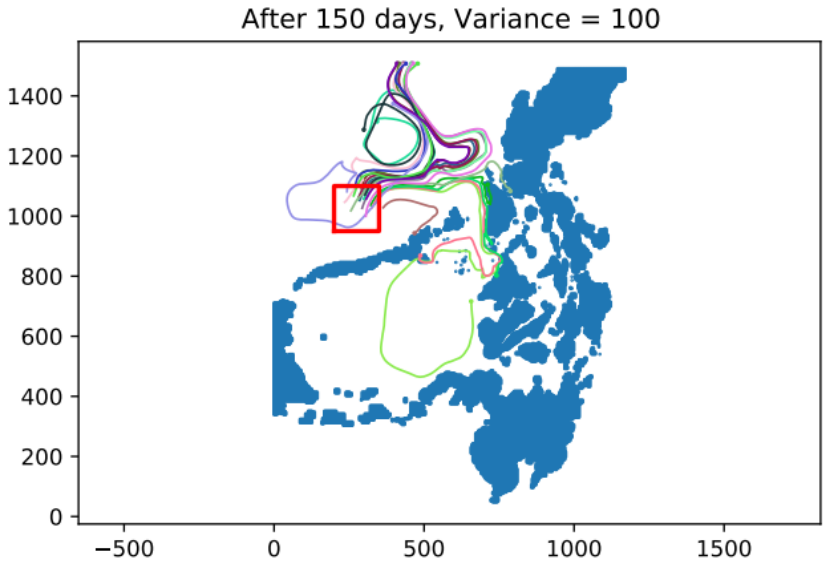


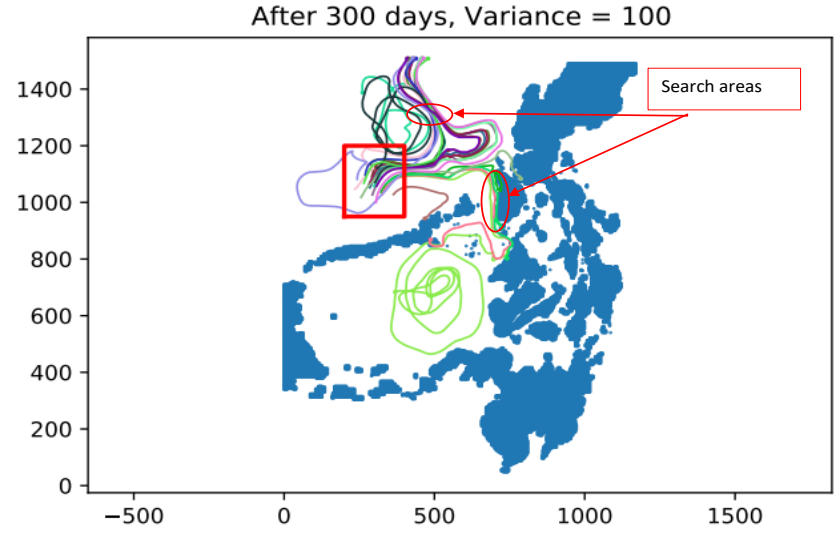
Sorry for poor representation, Google Colab cannot do better :)

As before, the blue points are seen observations connected with the red line. The green line is the prediction and shaded region is the 3σ confidence interval

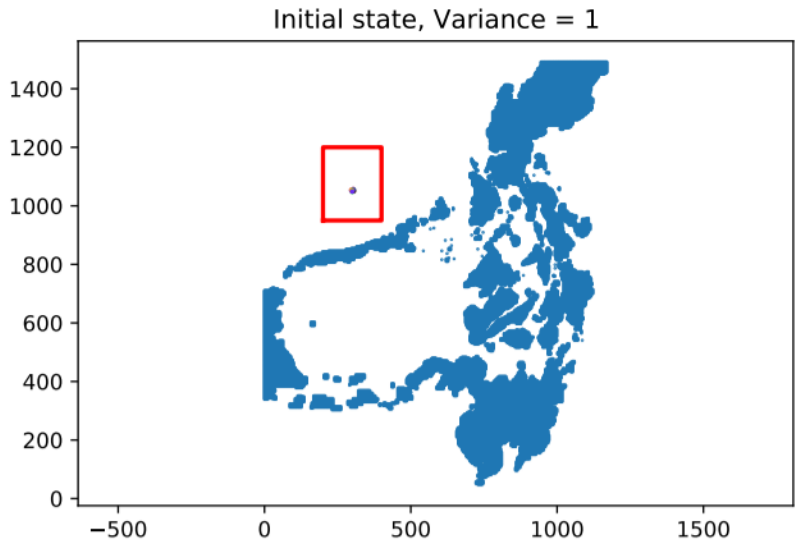
Problem 6

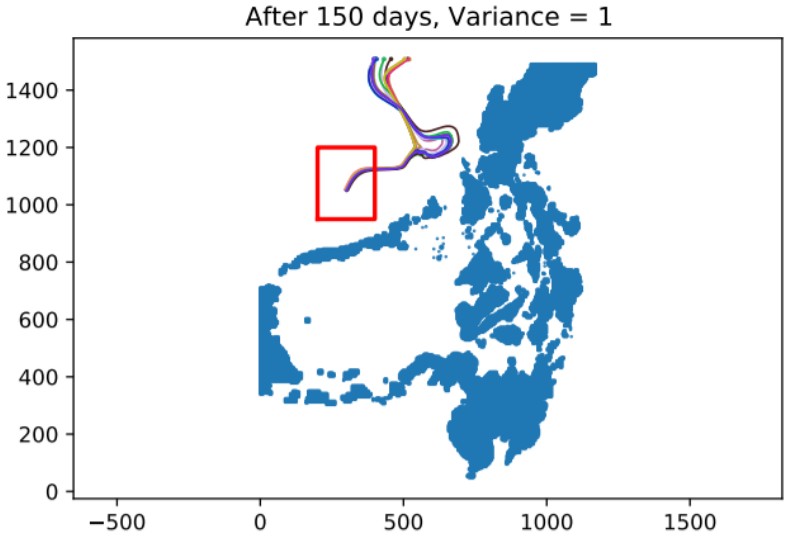

Initial state, Variance = 100
1400
1200
1000
800
600
400
200
0
−500
0
500
1000
1500

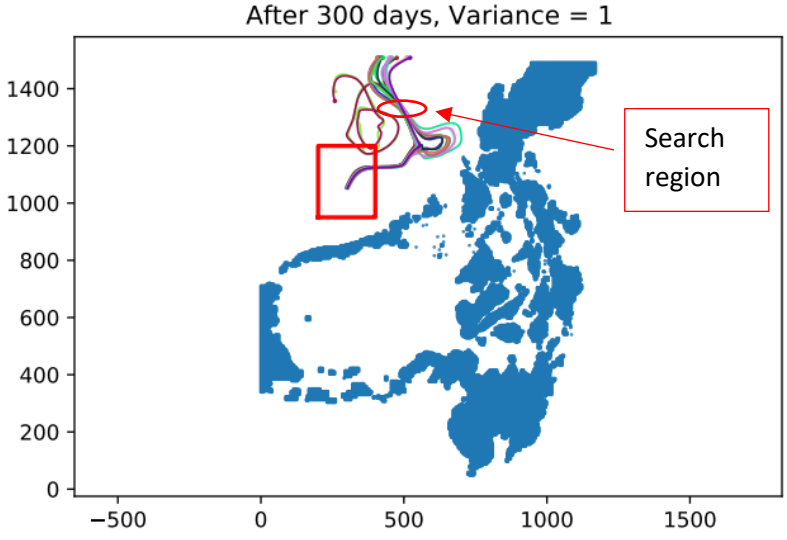

After 150 days, Variance = 100
1400
1200
1000
800
600
400
200
0
−500
0
500
1000
1500


After 300 days, Variance = 100
Search areas
1400
1200
1000
800
600
400
200
0
−500
0
500
1000
1500

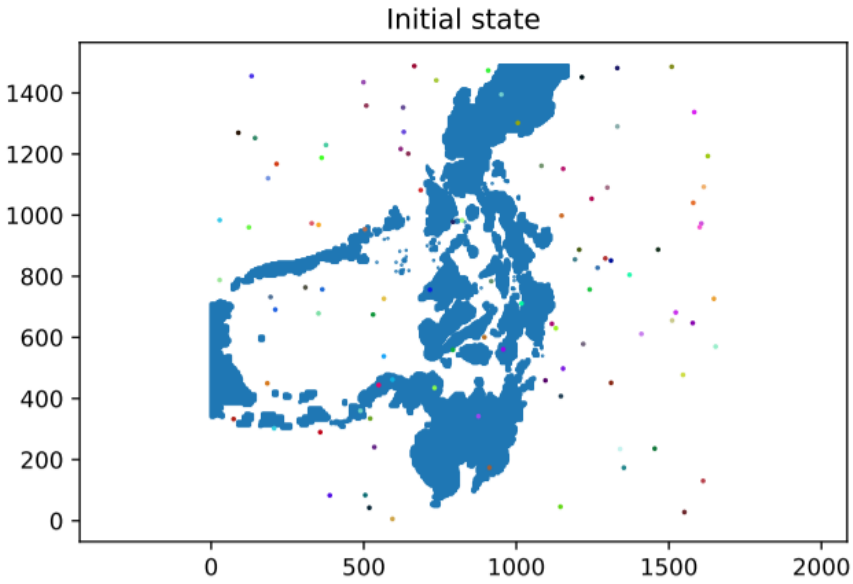
Var 100 means that the position of the plane crash is given with 20x20 km2 of 95% confidence area. The variance of 1 reduces this area down to 2x2 km2. The figures for the latter case are given below

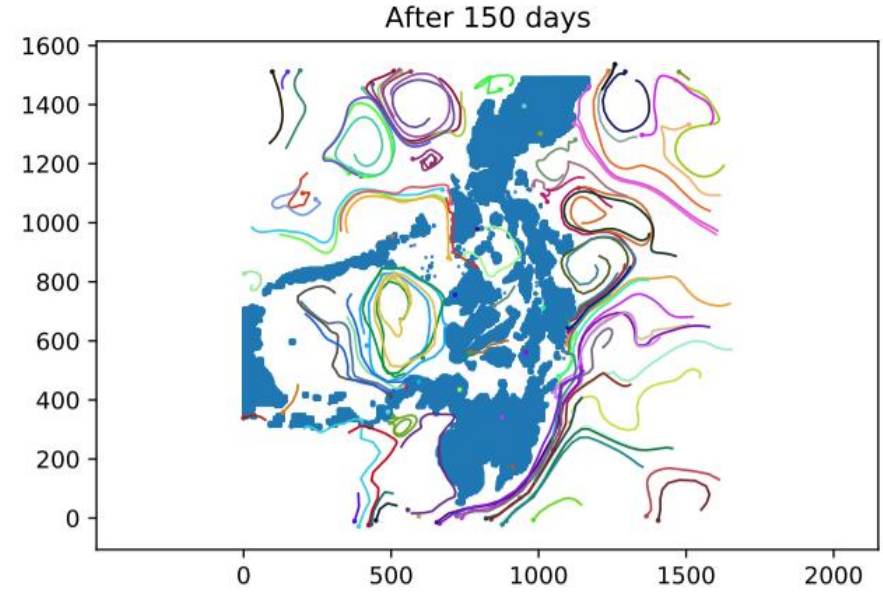


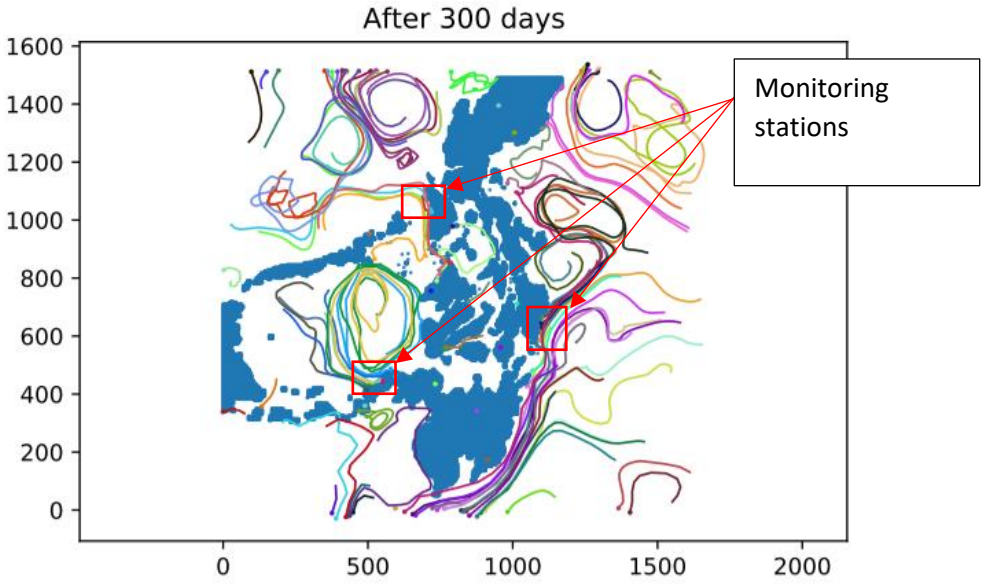




We can clearly mark one location in the ocean through which the majority of the debris pass, even in case the place of plane's crash is not known exactly – it is area around (500km, 1300km) in the coordinates of the above figures. Those debris which do not flew in Nord direction in the ocean, are generally passing along the west coast of Mindoro Island. But if the location of the plane is known with high confidence, the region in the ocean does not change, but become smaller in area, so that it makes sense to only concentrate the search efforts around (500km, 1300km) point. In case of variance equal to 1 no debris arrive to Mindoro







Sorry for the particles on land, please ignore them, I didn't have time to fix it unfortunately

For the selection of location of search, we look at the places where multiple points ended their journey or multiple paths lie close to the coast.