IDS.131Problem Set 3

Yash Dixit

Part I – Theory

Problem 4.1: Gaussian Processes

(a) Prediction with GPs

- Recap of key process steps in predictive GP algorithm:

$$\mathrm{cov}(Y_i, Y_j) = k(x_i, x_j) = \exp\Big(-\frac{\|x_i - x_j\|^2}{2\ell^2}\Big)$$

1. Pair-wise covariances: exp(squared distance)

$$\begin{bmatrix} \mathrm{cov}(Y_1, Y_1) & \dots & \mathrm{cov}(Y_1, Y_N) \\ \mathrm{cov}(Y_2, Y_1) & \dots & \vdots \\ \vdots & \ddots & \vdots \\ \mathrm{cov}(Y_N, Y_1) & \dots & \mathrm{cov}(x_N, x_N) \end{bmatrix} = \begin{bmatrix} k(x_1, x_1) & k(x_1, x_2) & \dots & k(x_1, x_N) \\ k(x_2, x_1) & k(x_2, x_2) & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ k(x_N, x_1) & \dots & & k(x_N, x_N) \end{bmatrix}$$

2. Covariance matrix (kernelled): NxN with each term given by exp(sq distance)

$p(y_*|y_{1:N})$ Gaussian with mean and variance

$$\mu_{*|1:N} = \mu_* + k_*^\top K_N^{-1} y_{1:N}$$
$$\sigma^2_{*|1:N} = \kappa_* - k_*^\top K_N^{-1} k_*$$

3. Predicted statistics entail inverting the covariance matrix

- As the number of observations increase, the computational complexity increases steeply due to first, the kernel operation for covariance matrix calculation and second, the matrix inversion step

- As the inversion step is characterized by $O(n^2)$, we can conclude that the prediction with GPs becomes expensive with a greater number of observations

(b) Increasing efficiency of this computation

- General idea:
  - Identify the computational bottleneck i.e. matrix to be inverted

- Approximate the matrix using 'shrinking' tools
- Implement inversion on the modified matrix making the process more efficient

- Factorize the covariance matrix using the Singular Value Decomposition
  - Dimension reduction from n to k
  - Preserve maximum information by selecting the significant principal components

- Sherman-Morrison-Woodbury

$$A_1 = \begin{bmatrix} A_{11} & U_1 V_1^T \\ V_1 U_1^T & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & 0 \\ 0 & A_{22} \end{bmatrix} + \begin{bmatrix} U_1 & 0 \\ 0 & V_1 \end{bmatrix} \begin{bmatrix} 0 & V_1^T \\ U_1^T & 0 \end{bmatrix}$$

$$(A+UV)^{-1} = A^{-1} - A^{-1}U\underbrace{\left(I + VA^{-1}U\right)^{-1}}_{P}VA^{-1}$$

If U, V are rank k then P is a k x k matrix i.e. computationally efficient

- We can thus conclude that the inverse computation is made more efficient

Part II – Ocean Flow

Problem 4.2: Flows and correlation

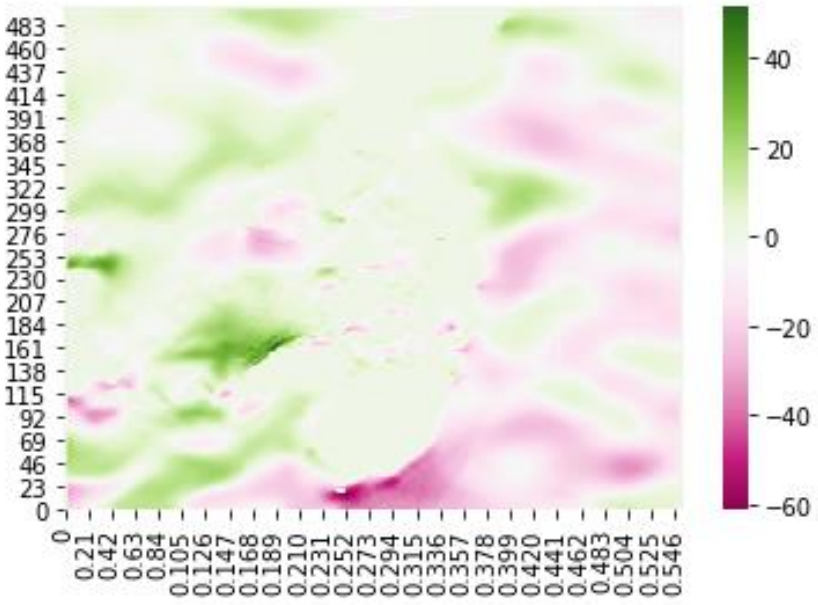
(a) Time averaged flow &
(b) Time averaged speed

- Snippets of the code used to compute time averaged velocities

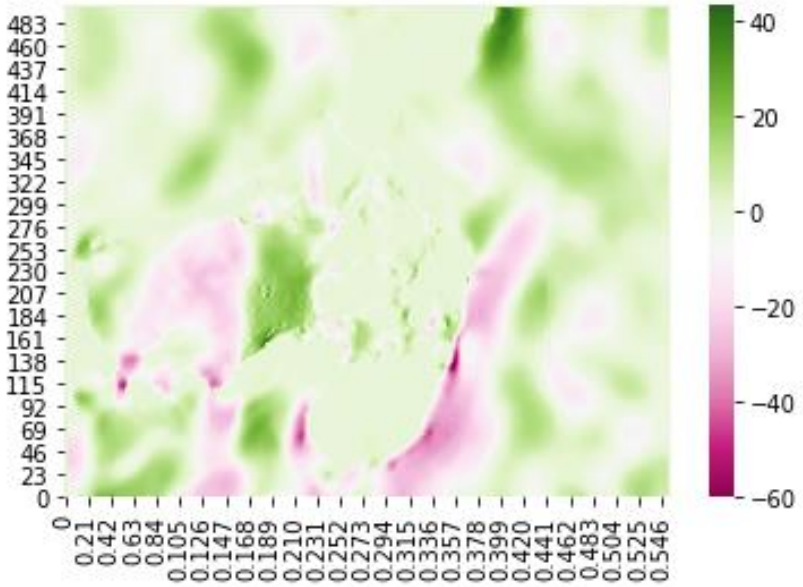
```
# Reading u, v flow data separately

USum = pd.read_csv("1u.csv", sep = ',')
USum[:] = 0

for uindex in range(1,101):
    name = "{index}u.csv".format(index = uindex)
    U = pd.read_csv(name, sep=',')
    U = U.multiply((25/0.9))
    USum = USum + U

UAvg = USum / 100
```

```
#uax = sns.heatmap(UAvg, cmap = "PiYG")
#uax.invert_yaxis()


VSum = pd.read_csv("1v.csv", sep = ',')
VSum[:] = 0

for vindex in range(1,101):
    name = "{index}v.csv".format(index = vindex)
    V = pd.read_csv(name, sep=',')
    V = V.multiply((25/0.9))
    VSum = VSum + V

VAvg = VSum / 100
#
#vax = sns.heatmap(VAvg, cmap = "PiYG")
#vax.invert_yaxis()

#plt.show()
```

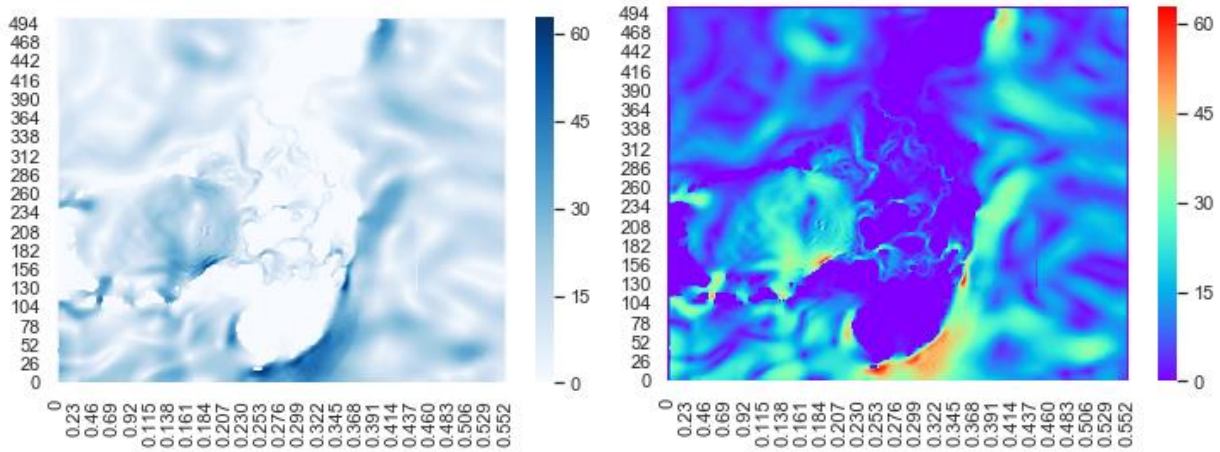
- Visualizations



Horizontal direction – Time averaged flow velocities



Vertical direction – Time averaged flow velocities



Visualization of the net velocity magnitude

- Snippets of the code used for finding velocity magnitude and direction

```
# Average velocity function
def avgvel(UX,UY):
    UXsq = UX.multiply(UX)
    UYsq = UY.multiply(UY)
    Velsq = UXsq + UYsq
    Vel = Velsq.apply(np.sqrt)
    return Vel

# Direction vector of flow field
def dirn(UX,UY):
```

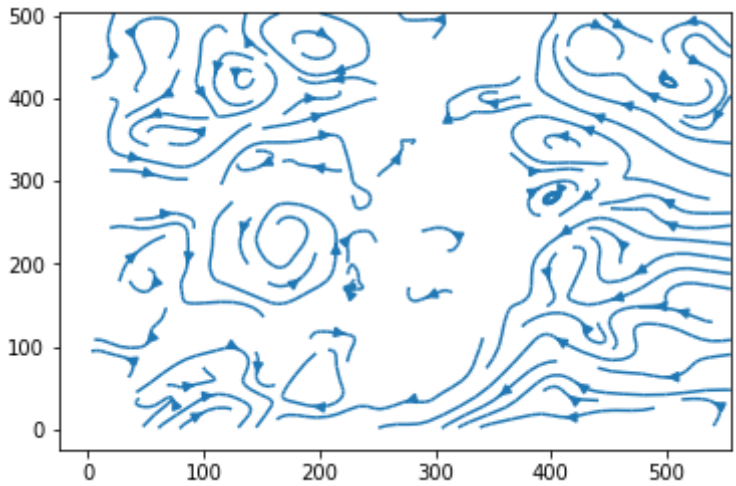
```
    ratio = UY.divide(UX)
    taninv = ratio.apply(np.arctan)
    return taninv
```

- Observations

  - Flow near the 'southern' part of the island appears relatively uniform with flow vectors aligned to the land contours (this is intuitive as ocean currents near land are shaped by the land contour)
  - Flow near the top left is uniform i.e. the ocean currents away from the land contours
  - Another approach to better visualize flow uniformity would be to compute gradients

```
for xin in range(1,555):
    for yin in range(1,503):

        Ugrad.iloc[yin:yin+1,xin:xin+1] =
UAvg.iloc[yin:yin+1,xin+1:xin+2] -
UAvg.iloc[yin:yin+1,xin:xin+1]

        Vgrad.iloc[yin:yin+1,xin:xin+1] =
VAvg.iloc[yin+1:yin+2,xin:xin+1] -
UAvg.iloc[yin:yin+1,xin:xin+1]
```

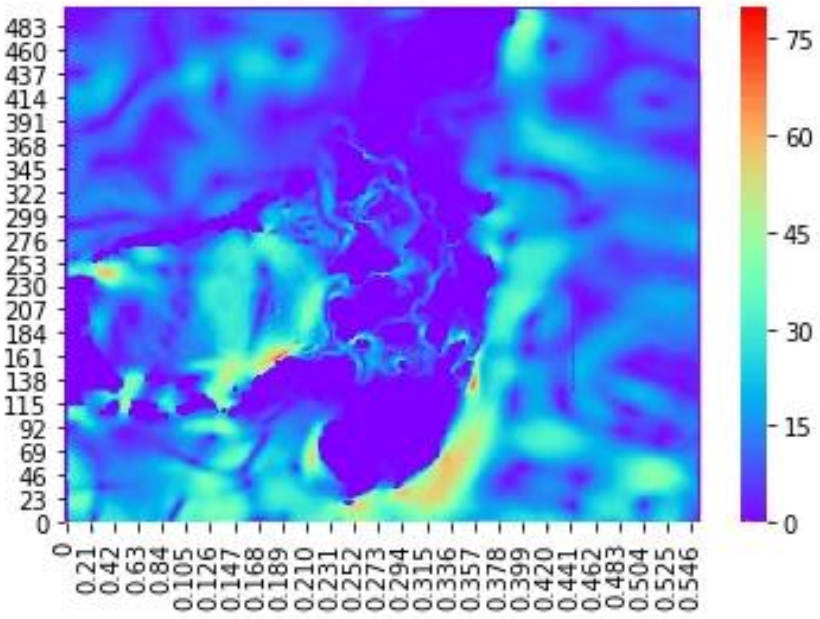
  - The average vector field too gives an accurate understanding of the uniform flow patterns as seen below



Average vector field visualization

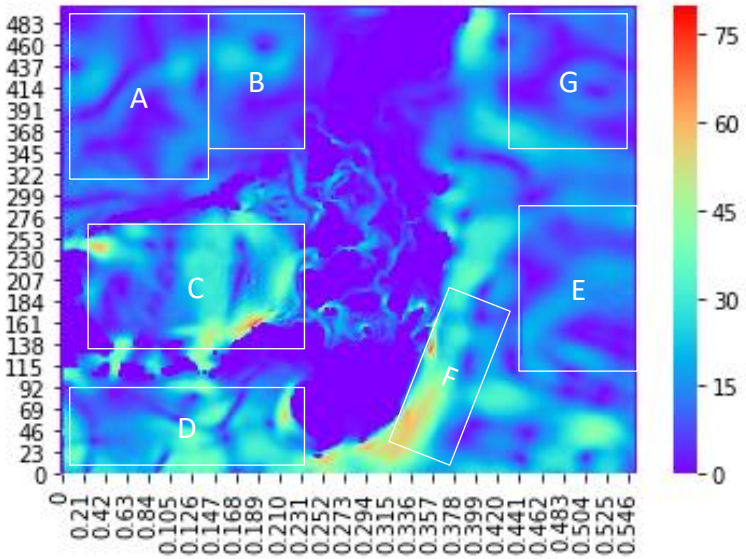
(c) Evolution of the flow with time

- Temporal flow visualization



(Double click on the above gif for the temporal visualization)

- Defining zones in the grid for better description of the spatial correlations



- We can observe distinctive spatial correlation within zones A-G

- The relationship *between* zones too is perceivable, although the directionality of the relationship isn't very explicit from the visualization

Problem 4.3: Predicting trajectories

(a) Particle trajectory prediction

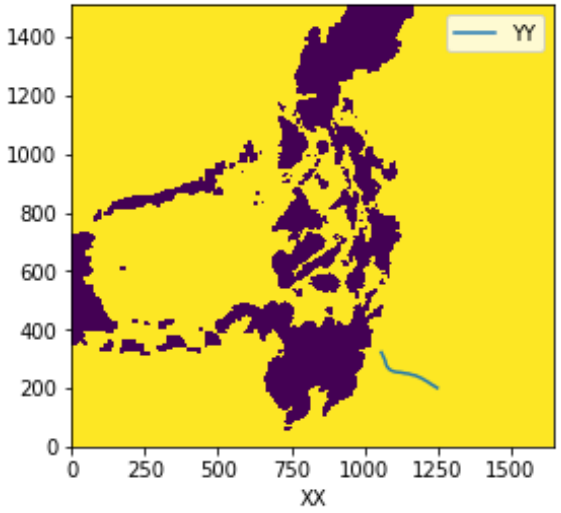
- Framework of the trajectory estimation algorithm
  - Initial location = Xi, Yi
  - Tag the grid point closest to the initial location = Xgi, Ygi
  - Reference the flow velocity Vi (at location Xgi, Ygi and at time t=0)
  - New location = Initial location + (Vi x Δt)
  - Iterate

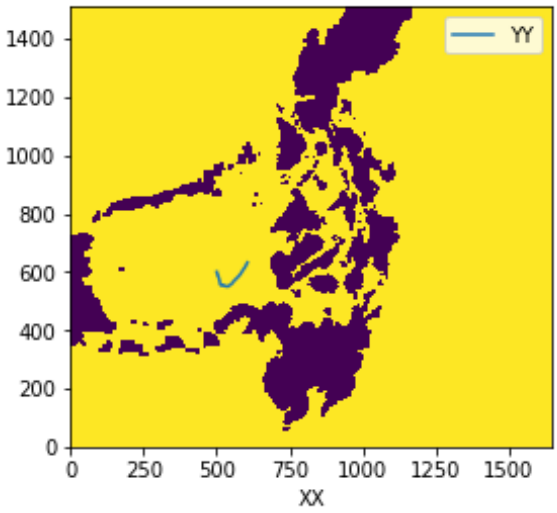
- Snippets of the code for trajectory prediction

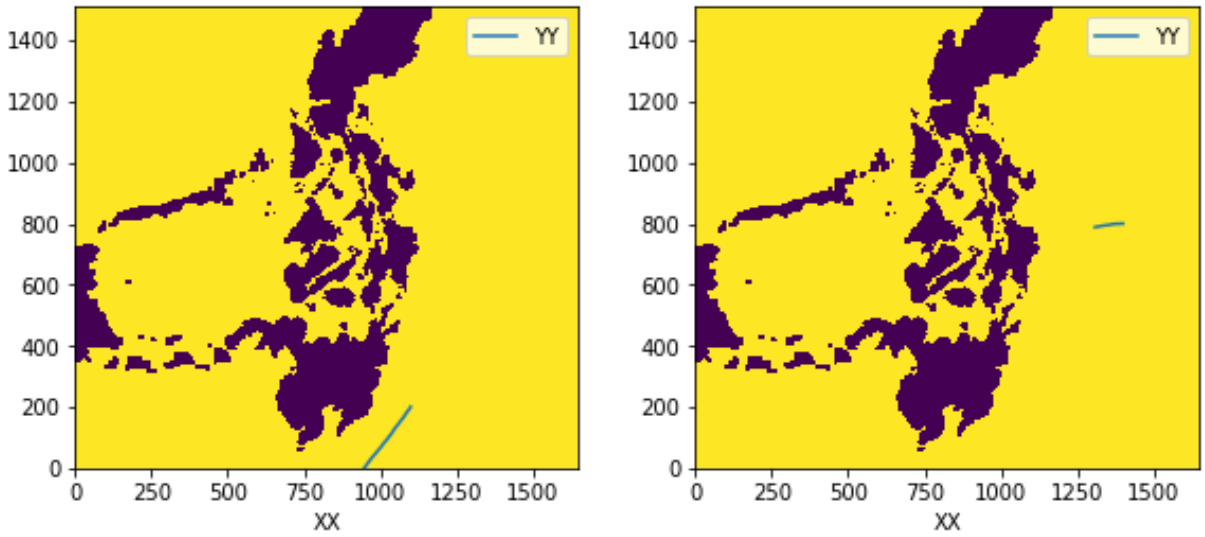
```
Xinitial = 100
Yinitial = 100

# Convert to grid units i.e. Xi'th and Yi'th position in the
grid matrix

Xi = Xinitial / 3
Yi = Yinitial / 3

Xi_grid = int(round(Xi))
Yi_grid = int(round(Yi))

# Create empty dataframe to store trajectory coordinates

col = ['XX','YY']
L = pd.DataFrame(index = range(101), columns = col)
L[:] = 0

L.loc[0] = [Xinitial,Yinitial]

Xnew = Xinitial
Ynew = Yinitial

for tindex in range(1,101):

    uname = "{index}u.csv".format(index = tindex)
    vname = "{index}v.csv".format(index = tindex)
    U = pd.read_csv(uname, sep=',')
    V = pd.read_csv(vname, sep=',')
    U = U.multiply((25/0.9))        # U in cm/sec
```

```
    V = V.multiply((25/0.9))        # V in cm/sec

    # Velocities at grid point

    UHolder = []
    VHolder = []

    UHolder = U.iloc[ Yi_grid : Yi_grid + 1 , Xi_grid :
Xi_grid + 1 ]
    uscalar = UHolder.values

    VHolder = V.iloc[ Yi_grid : Yi_grid + 1 , Xi_grid :
Xi_grid + 1 ]
    vscalar = VHolder.values

    # New trajectory coordinates

    Xnew = ( (Xnew) + ((uscalar*3*60*60)/(100*1000)) )
    Ynew = ( (Ynew) + ((vscalar*3*60*60)/(100*1000)) )

    L.loc[tindex] = [Xnew, Ynew]
```

- Visualizations







(b) Expected locations for the flow particle

- Snippets of the code (random draw from gaussian + terminating the trajectory prediction in the given timeframe)

```
#loc =
np.array([[200,450],[750,200],[1200,1350],[1500,600],[1500,800],[
500,600],[250,1200]])
loc = np.random.multivariate_normal([300,1050],[[50, 0], [0,
50]],10)
(m,n) = loc.shape

plt.figure()

# Initial location in km
for c in range(0,m):

    (Xinitial, Yinitial) = loc[c]

    stdev = 50
    # Convert to grid units i.e. Xi'th and Yi'th position in the
grid matrix

    Xi = Xinitial / 3
    Yi = Yinitial / 3

    Xi_grid = int(round(Xi))
    Yi_grid = int(round(Yi))

    # Create empty dataframe to store trajectory coordinates
```

```
time_scale = 40
col = ['XX','YY']
L = pd.DataFrame(index = range(time_scale), columns = col)
L[:] = 0

L.loc[0] = [Xinitial,Yinitial]

Xnew = Xinitial
Ynew = Yinitial

for tindex in range(1,time_scale):
  .
  .
  .
```

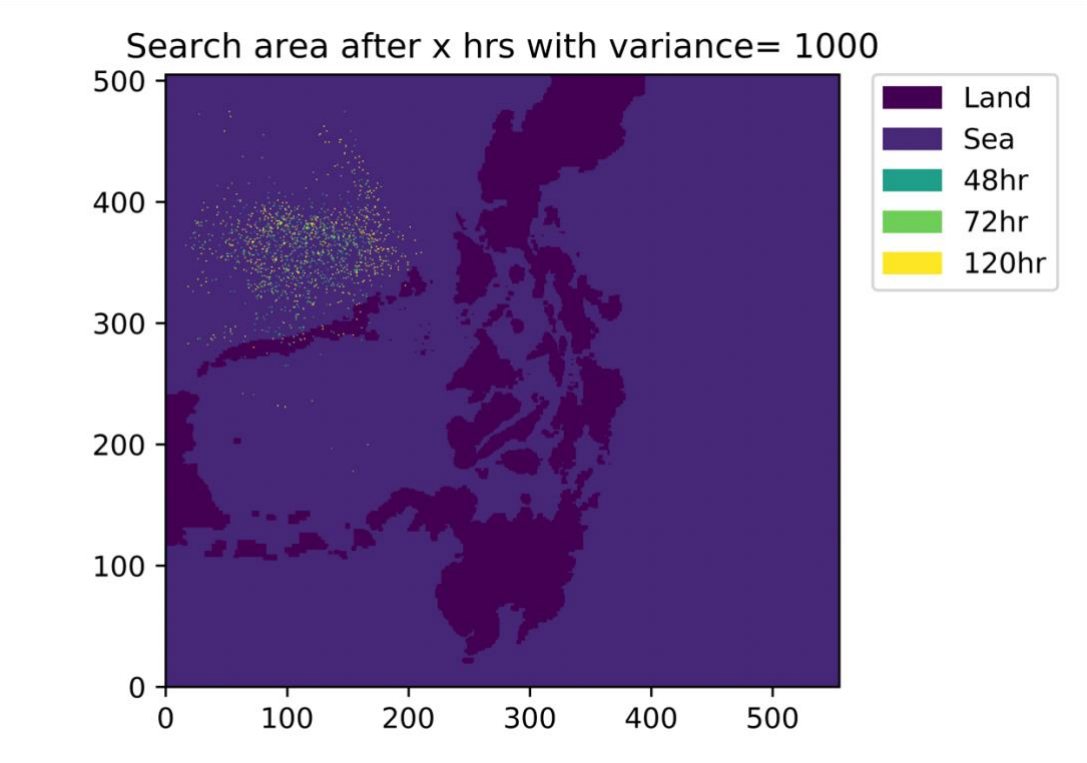
- Low variance (sigma = 50) – precise and narrow search area

T=48 hrs

T=72 hrs

T = 120 hrs

- High variance



- Observations

  - As the variance increases, the search area space becomes larger with greater levels of uncertainty.
  - There isn't much change in the center of the spread of search locations with the path areas seeming consistent in both graphs
  - The speed of the surrounding flows would also amplify the effects of the spread of search locations since intuitively, a small change in direction with higher flow speed would cause the potential search locations to be more spread out.

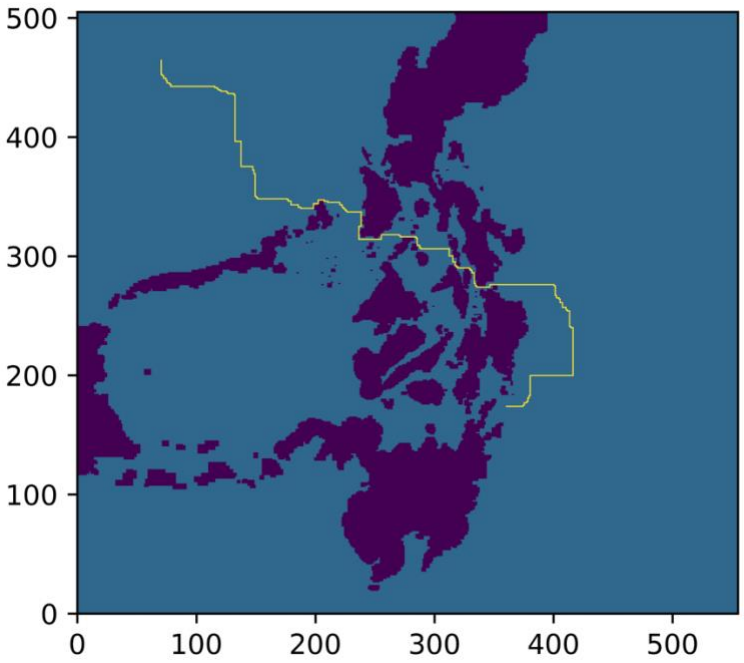
4.4: Path Planning

(a) Planning a route to minimize travel time

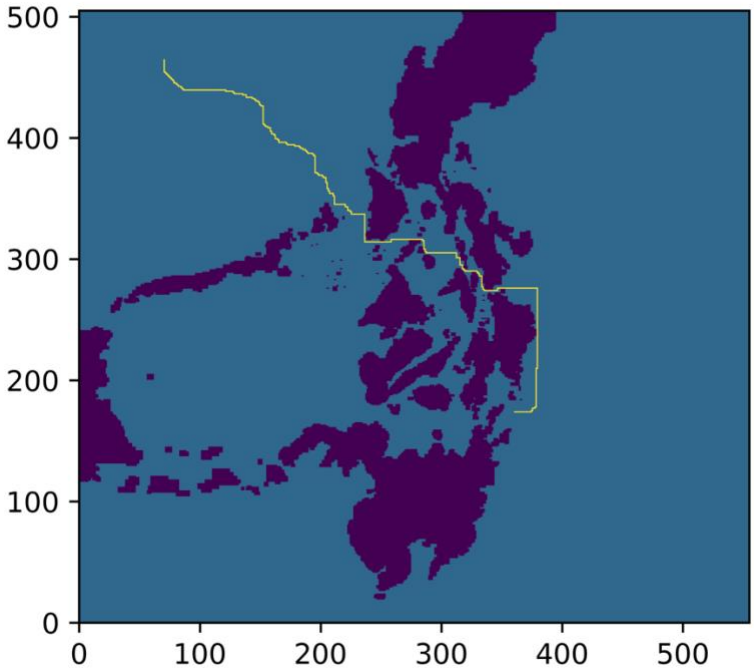
- Strategy
  - First, we will compute the velocity and hence time required for each point to reach its neighboring points, assuming that the points are not on land. For the simplify the problem, we will assume that the boat can only go in 4 directions, the North, South, East and West (not diagonally). The time taken was computed

using the distance from each grid (3km) divided by the speed in a particular direction with the velocity V included as part of the speed since the boat is equipped with the engine.

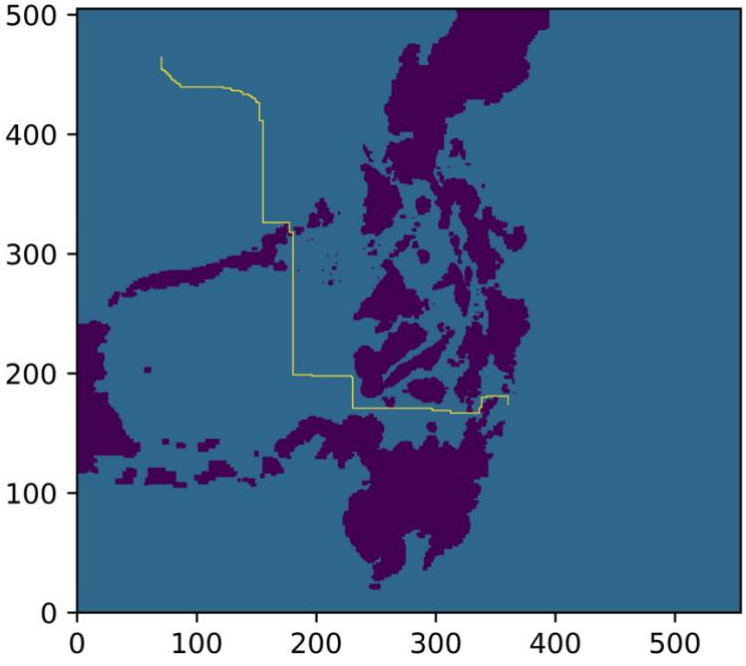
- We then construct a weighted, directed network with nodes as the points in the grid (there are no nodes created for land areas). Edges were then added between neighboring nodes (north, south, east, west) with the weight equal to the time taken to travel from the current node to each of its neighbors. To determine the shortest path, we run Dijkstra's algorithm, setting the source node as (x_0, y_0) = (70, 400) and the destination node as (x_f, y_f) = (360, 170).
- The shortest path was then determined for various V = 0.3, 3, 30 and 300km/hr and figures 10, 11, 12 , 13 depict the shortest paths respectively.



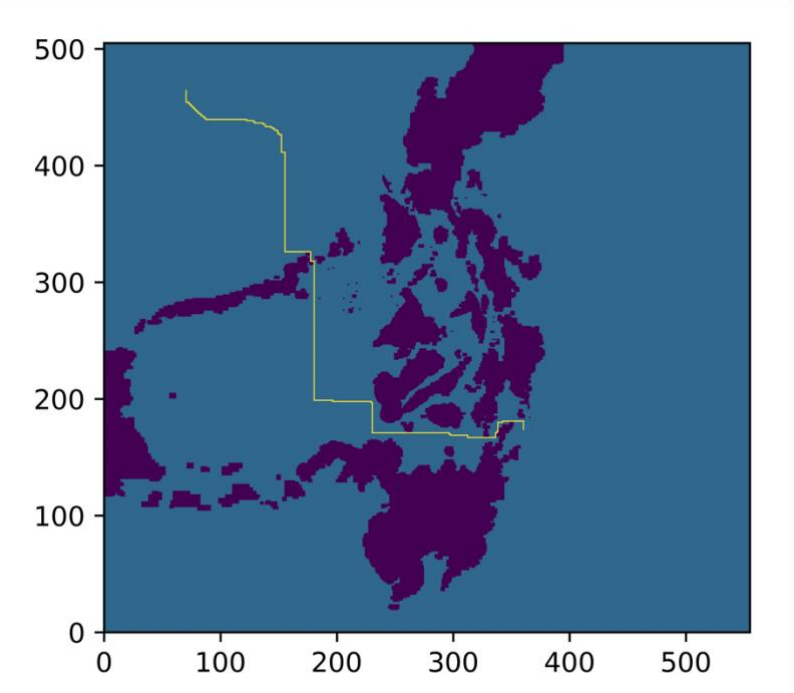
Shortest path with V = 0.3km/hr. Time taken: 3416.34hr



Shortest path with V = 3km/hr. Time taken:495.13hr



Shortest path with V = 30km/hr. Time taken: 51.29hr



Shortest path with V = 300km/hr. Time taken: 5.15hr

- Key observations

  - We see that the path for between 0.3km/hr and 3km/hr differ with the slower speed taking a longer path. This is most likely due to destruction of velocity components i.e. the slow speed not able to overcome the strong currents and thus the boat has to take an alternative path to mitigate this.
  - This can also explain the path difference when V increases to 30km/hr and 300km/hr where they take an entirely different path. The latter speeds have a similar path, which seem to indicate that once they overcome a certain threshold of the ocean flow, they are able to better optimize their path. The power of the engine of the boat directly translate to the velocity of the boat which also accounts for a large difference in time taken to complete the journey.

(b) Time varying flow

- To incorporate information of time varying flow, we can employ a similar set up with nodes and edges from 4(a) but deploy a greedy algorithm
- That is, we look to optimize the shortest path at each time step. Assuming we know the starting and ending node, we can do this using with the following greedy algorithm
  - At time = 1, we compute the time taken to reach the destination using the flow at time = 1, at the starting position and select the moving to the 1-neighbor (immediate) following the path computed. At the new position, at time = 2, we

will use that as a starting position and compute the shorted path to the destination with flow velocity information at time = 2. We will do this at every node at each time step until we reach our destination. Here we are optimizing for the shortest path at each time step and hence, we might not necessarily get a global optimal.