Map Matching Benchmarks

This notebook is intended as a supplement to the Sendai Map notebook. This notebook implements the competing map matching algorithms and tests them against the map-matching-dataset for comparison.

In [2]:

```
# Data Input

df_track = db.read_text('map-matching-dataset/*track.geojson').map(json.loads
df_network_edges = db.read_text('map-matching-dataset/*arcs.geojson').map(jsor
df_network_nodes = db.read_text('map-matching-dataset/*nodes.geojson').map(js
df_gt = db.read_text('map-matching-dataset/*route.geojson').map(json.loads).m
```

In our case, our data is already fused. But often you will have several datasets with asynchronous data that you will have to fuse first. We implemented a barebones method in mm_utils to handle this; you can see an example of how to use it in '02a Data Pre-Processing'.

In our case, our data is already fused. But often you will have several datasets with asynchronous data that you will have to fuse first. We implemented a barebones method in mm_utils to handle this; you can see an example of how to use it in '02a Data Pre-Processing'.

Typically GPS/IMU data is recorded as Point geometries in a GDF. However, some algorithms require a trajectory (LineStrings) despite this. As a result, our framework requires both points (nodes) and trajectories (edges). So we will need to create a "trajectory" by sequentially connecting our nodes

In our case, our data is already fused. But often you will have several datasets with asynchronous data that you will have to fuse first. We implemented a barebones method in mm_utils to handle this; you can see an example of how to use it in '02a Data Pre-Processing'.

Typically GPS/IMU data is recorded as Point geometries in a GDF. However, some algorithms require a trajectory (LineStrings) despite this. As a result, our framework requires both points (nodes) and trajectories (edges). So we will need to create a "trajectory" by sequentially connecting our nodes

In [3]:
```
# I wrote a utility function to do that, provided in mm_utils.

df_track_edges = df_track.map(mm_utils.point_to_traj)#, columns = {'timestamp
                                                       #  'altitude':'average',
                                                       #  'speed':'average',
                                                       #  'vertical accuracy':'
                                                       #  'horizontal accuracy'
                                                       #  'oops':'notavalidmeth
```

Now we demonstrate how we can work with several algorithms at once in a modular fashion.

First we initialize the simulators, to be later applied.

In [5]:

```python
from algorithms import fmm_bin
from fmm import FastMapMatchConfig

### Define map matching configurations

k = 8
radius = 0.003
gps_error = 0.0005

fmm_config = FastMapMatchConfig(k,radius,gps_error)
cfg_file = None


### Algorithm Implementation here, if not in separate .py file


###

sim1 = fmm_bin.FMM(cfg = fmm_config)
#sim2 = EKF.Sim(cfg = cfg_file)


## If you have the ground truth, load it here
ground_truth = db.read_text('map-matching-dataset/*route.geojson').map(json.l
```

In [6]:

```
## We will convert our Dask Bags to Dask Delayed objects so we can iterate ove
# Finally we will compute our results, and Dask will automatically paralleliz

gt = ground_truth.to_delayed()
te = df_track_edges.to_delayed()
tn = df_track.to_delayed()
ne = df_network_edges.to_delayed()
nn = df_network_nodes.to_delayed()
```

Now we are ready to run the simulator on a subsection (or all of) the data. Notice how easy it is to run algorithms in parallel-- all of the parallelization is handled by Dask Delayed, so even though our algorithm is only designed to handle one case at a time, it has already gained magnitudes of efficiency.

One small caveat-- even if your algorithm is technically 'independent', if you utilize python.os functions, you may run into I/O read/write errors. To circumvent this, use a dedicated library such as tempfile to systematically handle the temp file creation.

In [7]:
```python
## Let's see this!
sim1_results = []

n = 10

for i in range(n):
    sim1_results.append(dask.delayed(sim1.run)(te[i],
                                               tn[i],
                                               ne[i],
                                               nn[i],
                                               return_results=True))

from dask.distributed import Client, LocalCluster



cluster = LocalCluster()  # Launches a scheduler and workers locally
client = Client(cluster)  # Connect to distributed cluster and override defau
client.restart()
# This is the best scheduler to use when doing I/O operations (which FMM requ


sim1_results[:3]
sim1_results = dask.compute(*sim1_results, scheduler='sync')

#sim1.run(input_nodes = df_nodes, input_edges = df_edges, network_nodes = netw
#sim1.results
#sim1_results
```

```
2022-07-15 00:18:23,895 - distributed.diskutils - INFO - Found stale
```

Now we iterate through our results, and evaluate it using the build-in evaluation method in mm_utils.

In [43]:

```
errors = []
for i in range(len(sim1_results)):
#    errors.append(dask.delayed(mm_utils.evaluate)(sim1_results[i],gt[i], mat
# For whatever reason, delayed isn't working here, so I'm just going to do th
    errors.append(mm_utils.evaluate(sim1_results[i],
                                    gt[i].compute()[0],
                                    matchid = "index")) # A more standard mat

#errors = [dask.compute(*errors)]

#sim2.run(input_nodes = df_nodes, input_edges = df_edges, network_nodes = net
#pred_nodes, pred_edges = sim2.results
# I did implement plot() and evaluate() for sim2
# But for demonstration, I will use mm_utils, as some programmers may choose
#sim2.plot()
#sim2.evaluate()
#mm_utils.plot(network, df_nodes.append(df_edges), sim2.results())
#mm_utils.evaluate(sim2.results(), ground_truth, match='index'
```

In [49]:

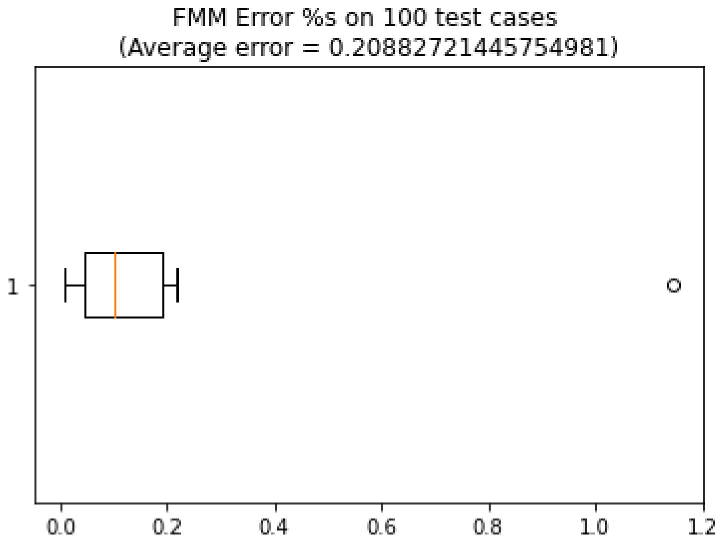
```python
fig = plt.figure(1)
plt.boxplot(errors,vert=False)
plt.title('FMM Error %s on '
          + str(n)
          + ' test cases \n(Average error = '
          + str(np.average(errors)) + ')')
plt.show()
```

Out[49]:

```
{'whiskers': [<matplotlib.lines.Line2D at 0x7ff875d6a740>,
  <matplotlib.lines.Line2D at 0x7ff875d6aa10>],
 'caps': [<matplotlib.lines.Line2D at 0x7ff875d6ace0>,
  <matplotlib.lines.Line2D at 0x7ff875d6afb0>],
 'boxes': [<matplotlib.lines.Line2D at 0x7ff875d6a470>],
 'medians': [<matplotlib.lines.Line2D at 0x7ff875d6b280>],
 'fliers': [<matplotlib.lines.Line2D at 0x7ff875d6b3d0>],
 'means': []}
```

Out[49]:

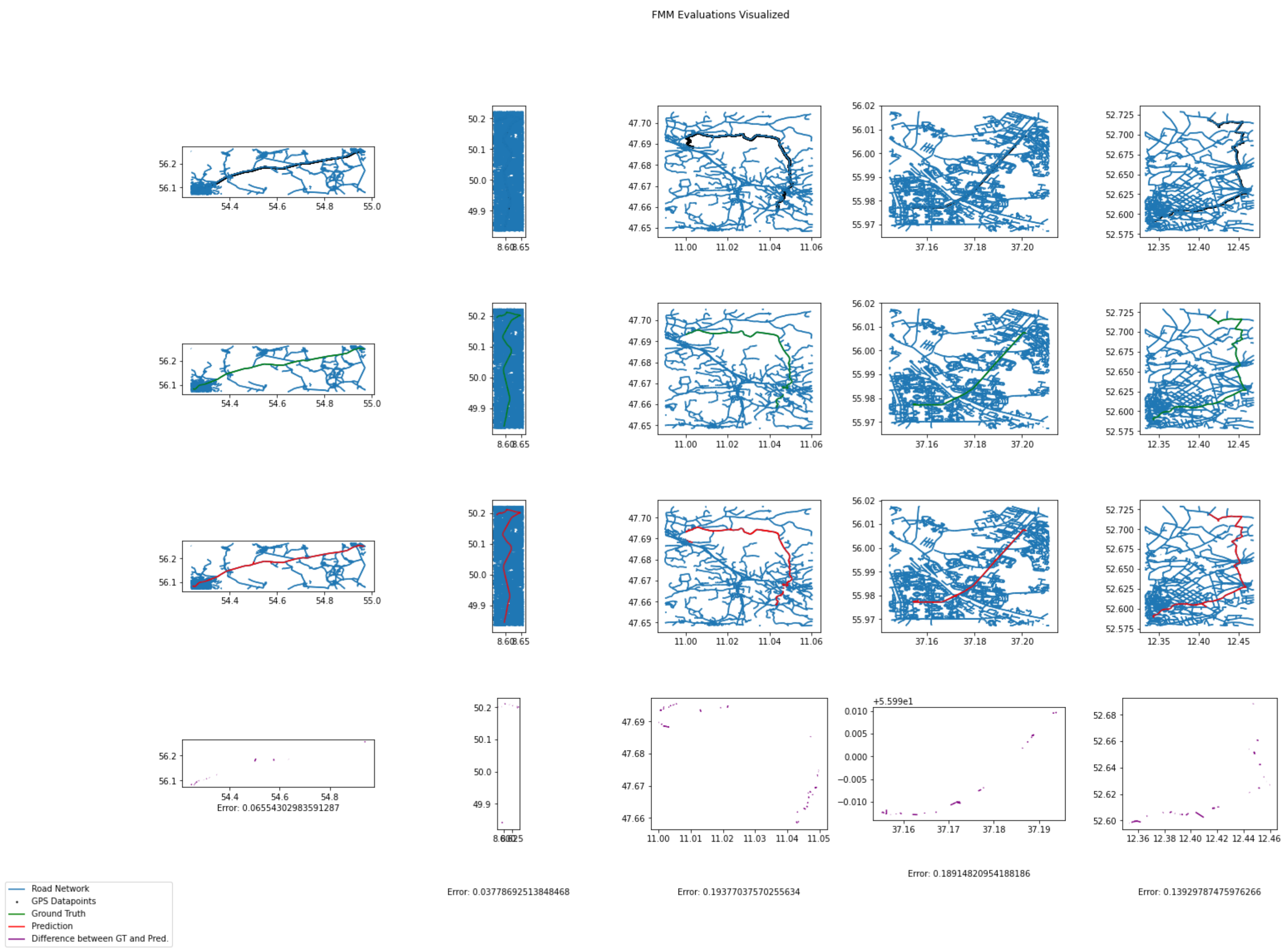
```
Text(0.5, 1.0, 'FMM Error %s on 100 test cases \n(Average error = 0.2
0882721445754981)')
```


FMM Error %s on 100 test cases
(Average error = 0.20882721445754981)
1
0.0
0.2
0.4
0.6
0.8
1.0
1.2

Overall, FMM seems to handle itself okay, with the exception of one huge outlier!

(Note-- having an error above 100% is not a bug; the standard error formula is not upper-bounded, despite being a percentage)

Let's see a few examples of our results.


FMM Evaluations Visualized
Error: 0.06554302983591287
Error: 0.03778692513848468
Error: 0.19377037570255634
Error: 0.18914820954188186
Error: 0.13929787475976266
Road Network
GPS Datapoints
Ground Truth
Prediction
Difference between GT and Pred.