

Viewing AIS data

This [Jupyter](https://jupyter.org) (<https://jupyter.org>) notebook explores and analyzes the [Automatic Identification System \(AIS\)](https://en.wikipedia.org/wiki/Automatic_identification_system) (https://en.wikipedia.org/wiki/Automatic_identification_system) vessel-location data provided for the 12/2020 VAULT technical scenario, showing what data is available and how it can be accessed and visualized from Python. The notebook also acts as a runnable application that can be put on a server to allow users to explore the data interactively.

```
In [1]: import pandas as pd
import numpy as np
import panel as pn
import datetime as dt
import holoviews as hv
import colorcet as cc
import param
from holoviews.util.transform import lon_lat_to_easting_northing as ll2en
from holoviews.operation.datashader import rasterize, dynspread
hv.extension('bokeh')
```



Data has been provided for four [UTM zones](https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system#UTM_zone) (https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system#UTM_zone) (1, 2, 3, and 10), with vessels identified by their [Maritime Mobile Service Identity](https://en.wikipedia.org/wiki/Maritime_Mobile_Service_Identity) (https://en.wikipedia.org/wiki/Maritime_Mobile_Service_Identity) numbers.

Here we will use [pandas](https://pandas.pydata.org) (<https://pandas.pydata.org>) to load 2017 data from zones 1-3 and 2014 data from zone 10:

```
In [2]: zone1 = pd.read_csv('./data/vessel_data/AIS/AIS_2017_01_Zone01.csv', parse_dates=[1])
zone2 = pd.read_csv('./data/vessel_data/AIS/AIS_2017_01_Zone02.csv', parse_dates=[1])
zone3 = pd.read_csv('./data/vessel_data/AIS/AIS_2017_01_Zone03.csv', parse_dates=[1])
zone10 = pd.read_csv('./data/vessel_data/Cleaned AIS/Zone10_2014_01/Broadcast.csv', parse_dates=[1])
```

Here the zone 1-3 data was provided directly as CSVs, while the zone 10 data was created from the provided GDB files using `ogr2ogr . ogr2ogr` produces several CSVs per GDB file, and here we focus only on Broadcast.csv. The column names for the zone 10 data are slightly different from the others, so we will first map from those to the format from the original csvs:

```
In [3]: columnmap = dict(mmsi_id='MMSI', date_time='BaseDateTime', lat='LAT', lon='LON', speed_over_ground='SOG',
                        course_over_ground='COG', heading='Heading', status='Status')

zone10.columns = [columnmap.get(c,c) for c in zone10.columns]
```

The AIS files include a variety of data and metadata values about each reported AIS broadcast. As we can see by looking at the first few rows of data, some of these values indicate the current state (e.g. LAT, LON, SOG, COG, Heading at the given time), while the others indicate metadata about the vessel (VesselName, CallSign, etc.):

```
In [4]: zones = pd.concat([zone1, zone2, zone3, zone10])
zones.tail()
```

```
Out[4]:
```

	MMSI	BaseDateTime	LAT	LON	SOG	COG	Heading	VesselName	IMO	CallSign	VesselType	Status	Length	Width
23633230	367445507	2014-01-31 23:59:36	47.922778	-122.684650	0.0	53.299999	511.0	NaN	NaN	NaN	NaN	0	NaN	NaN
23633231	338513412	2014-01-31 23:59:36	36.805180	-121.785770	0.0	183.000000	511.0	NaN	NaN	NaN	NaN	0	NaN	NaN
23633232	367097048	2014-01-31 23:59:47	44.655612	-124.337907	6.8	104.000000	511.0	NaN	NaN	NaN	NaN	0	NaN	NaN
23633233	367041647	2014-01-31 23:59:45	45.559840	-124.529237	9.5	198.500000	193.0	NaN	NaN	NaN	NaN	0	NaN	NaN
23633234	367870341	2014-01-31 23:59:49	41.234110	-124.460098	1.1	152.800000	511.0	NaN	NaN	NaN	NaN	0	NaN	NaN

Together this set of four files has more than 20 million records (AIS pings). Not all columns are present for every record, but some could be filled in using publicly available data (or replaced with a placeholder, such as using MMSI for a missing (NaN) VesselName).

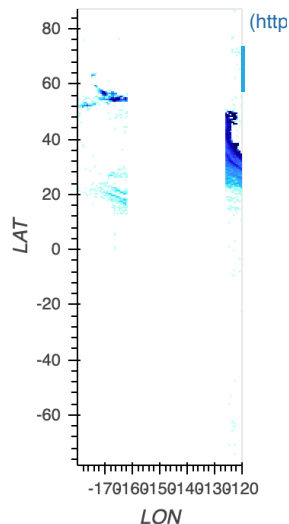
Because the eventual task involves matching vessel locations to satellite fields of view, let's visualize the set of locations available using [HoloViews](https://holoviews.org) (<https://holoviews.org>):

```
In [5]: print(f"Ranges found: latitude {min(zones.LAT):.4} to {max(zones.LAT):.4}, "
          f"longitude {min(zones.LON):.4} to {max(zones.LON):.4}")
```

Ranges found: latitude -78.22 to 87.15, longitude -180.0 to -120.0

```
In [6]: points = rasterize(hv.Points(zones, ['LON', 'LAT'])).opts(cnorm='eq_hist', aspect='equal')
points
```

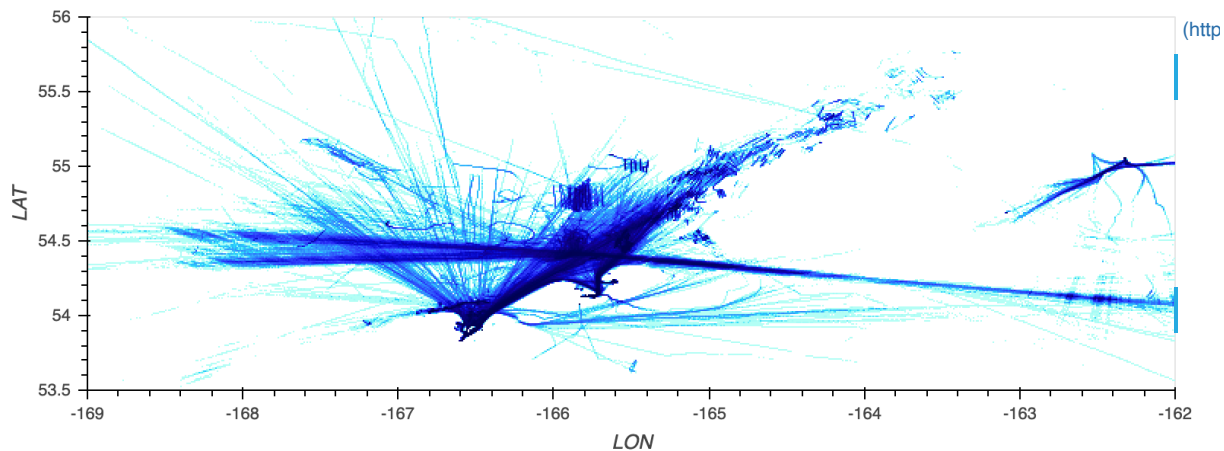
Out[6]:



Here HoloViews generates an interactive [Bokeh](https://bokeh.org) (<https://bokeh.org>) plot that uses [Datashader](https://datashader.org) (<https://datashader.org>) to compute a heatmap for location data, with darker blue colors indicating pixels with more AIS data "pings". With an interactive Python session running, you can zoom into the above heatmap plot to see how the location data is distributed. E.g. if we zoom into the densest region of AIS points in the top left, we can see a lot of interesting structure:

```
In [7]: points = hv.Points(zones, ['LON', 'LAT'])
dyspread(rasterize(points, width=800, height=300, x_range=(-169,-162), y_range=(53.5,56))\
          .opts(width=800, height=300, cnorm='eq_hist', tools=['hover']))
```

Out[7]:



To put this data in context, let's plot it on a map. We'll use a public map tile source in Web Mercator coordinates, so we'll first project the LON,LAT coordinates to easting,northing values:

```
In [8]: %%time
zones.loc[:, 'x'], zones.loc[:, 'y'] = ll2en(zones.LON,zones.LAT)
```

CPU times: user 3.37 s, sys: 2.38 s, total: 5.75 s
Wall time: 5.75 s

Let's also create some annotations that show the UTM zone boundaries, to validate that the data provided is indeed in the zones listed in the filenames:

```
In [9]: def zone(i):
        """
        Return plottable bounds object for a given UTM zone
        (see https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system#UTM_zone)
        """
        m = l12en((-180+6*(i-1), -180+6*i), (-80, 84))
        bnds = hv.Bounds((m[0][0], m[1][0], m[0][1], m[1][1])).opts(color="white")
        text = hv.Text(m[0][0] + (m[0][1]-m[0][0])/2, 0, f"{i}").opts(color="white", text_font_size="5pt")
        return bnds * text
```

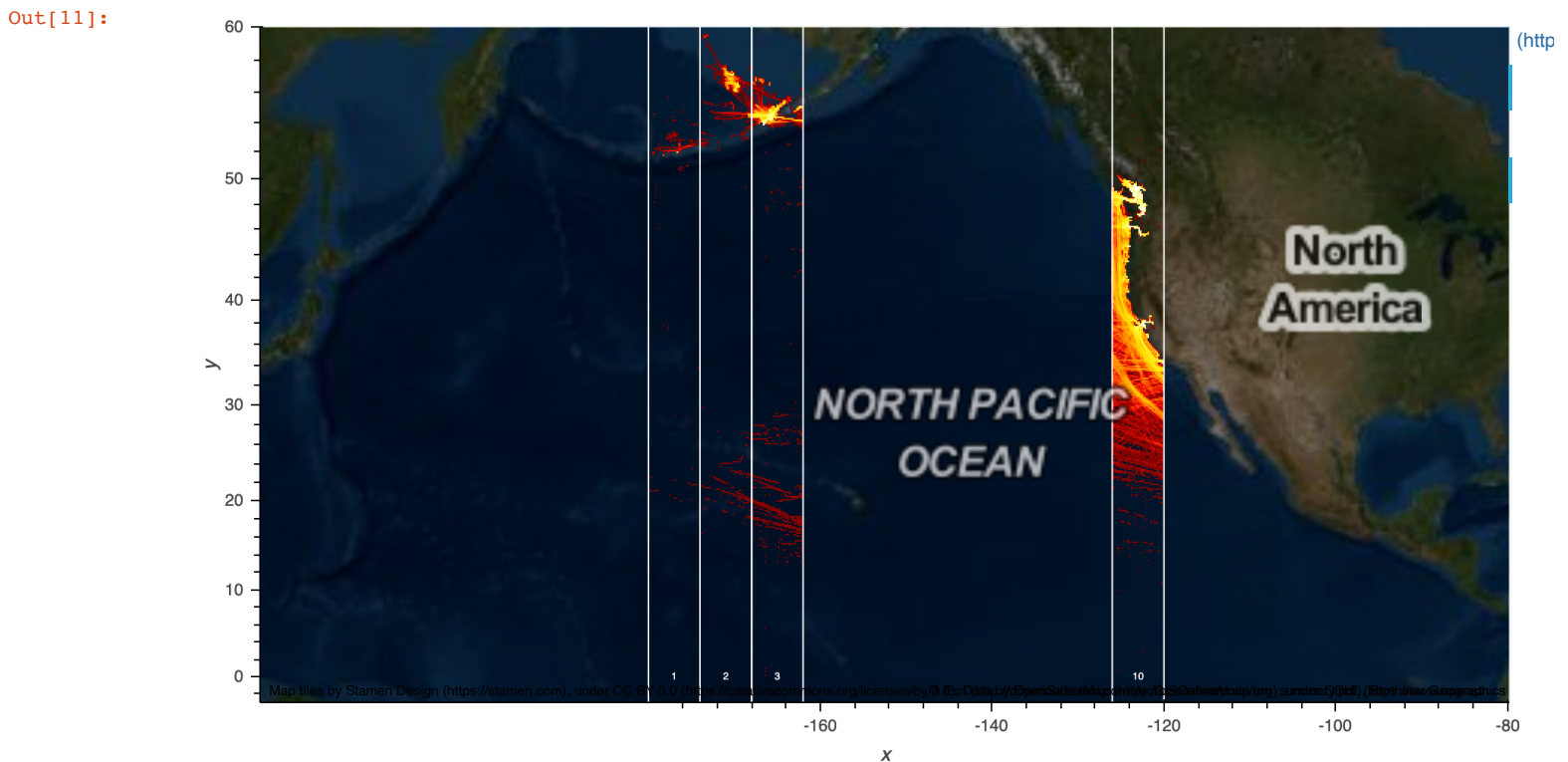
And let's set up some defaults, including a suitable lon,lat range and a colormap going from red to yellow to white so that it shows up against the dark ocean:

```
In [10]: x_range, y_range = l12en([-180-45, -80], [-3, 60])
        bounds = dict(x=tuple(x_range), y=tuple(y_range))
        opts = hv.opts.Image(cmap=cc.fire[64:], width=900, height=500, cnorm='eq_hist', alpha=1)
```

We can now plot the data on top of a map, with bounding boxes for the UTM zones, using `*` to overlay each item:

```
In [11]: points = rasterize(hv.Points(zones, ['x', 'y']).redim.range(**bounds)).opts(opts)
        tiles = hv.element.tiles.EsriImagery().opts(alpha=0.5, bgcolor='black')
        labels = hv.element.tiles.StamenLabels().opts(alpha=0.7, level='annotation')

        tiles * dynspread(points) * zone(1) * zone(2) * zone(3) * zone(10) * labels
```



Zooming and panning (selecting appropriate tools from the plot toolbar if needed) should reveal that these vessels provide many AIS pings around ports and in shipping lanes, as well as revealing other interesting trajectories and movement patterns.

Selecting a vessel at a given time

The above plots show the cumulative AIS location data over all times available in the files. If we are given a *particular* time, we can overlay markers for each vessel on top of the cumulative location data, showing the location of that vessel at the given time. There can be large gaps in the AIS data, so we interpolate between the most recent and the next AIS pings for the given time.

```
In [12]: vessels = {name:df.drop_duplicates().sort_values(by='BaseDateTime').set_index('BaseDateTime')}
        for name, df in zones.groupby('VesselName')}
        columns = list([el for el in zones.columns if el != 'BaseDateTime'])
```

```

In [13]: def interpolate_position(time, match_start, match_end):
    slon, slat = match_start['LON'], match_start['LAT']
    elon, elat = match_end['LON'], match_end['LAT']

    delta = match_end.name - match_start.name
    interval = time - match_start.name
    ratio = (interval / delta)
    lon_delta = (elon - slon) * ratio
    lat_delta = (elat - slat) * ratio
    mlon, mlat = (slon + lon_delta), (slat + lat_delta)
    return (slon, slat), (mlon, mlat), (elon, elat)

def vessel_at_time(vessel_name, time, vessels):
    df = vessels[vessel_name].drop_duplicates()
    if (time < df.index[0]) or (time > df.index[-1]):
        return None, None, None, None # Query before first or after last value
    try:
        idx = df.index.get_loc(time, method='ffill')
        match_start = df.iloc[idx]
        try:
            match_end = df.iloc[idx+1] # df.index.get_loc(time, method='bfill')
        except:
            match_end = match_start
        spos, mpos, epos = interpolate_position(time, match_start, match_end)
        return spos, mpos, epos, match_start
    except Exception as e:
        return None, None, None, None

marked_points = None # TODO: Declare a class and make this an attribute
def mark_vessels(value, show_segments=False):
    """Mark vessel location along with optional gap display"""
    global marked_points
    records = []
    segment_data1 = []
    segment_data2 = []
    empty = dict({'x':0., 'y':0., 'time':''}, **{col:'' for col in columns})
    for vessel in list(vessels.keys()):
        spos, mpos, epos, match = vessel_at_time(vessel, value, vessels)
        if match is not None:
            mx, my = ll2en(mpos[0], mpos[1])
            info = dict({'LON':mpos[0], 'LAT':mpos[1]}, **{col:match[col] for col in columns})
            records.append(dict(info, **{'x':mx, 'y':my, 'time':match.name}))
            sx, sy = ll2en(spos[0], spos[1])
            ex, ey = ll2en(epos[0], epos[1])
            segment_data1.append((sx,sy,mx,my))
            segment_data2.append((mx,my,ex,ey))
    markers = pd.DataFrame(records if len(records) != 0 else [empty])
    alpha = 1 if len(records) else 0
    segments1 = hv.Segments(np.array(segment_data1)) if len(segment_data1) else hv.Segments([(0,0,0,0)])
    segments2 = hv.Segments(np.array(segment_data2)) if len(segment_data2) else hv.Segments([(0,0,0,0)])
    marked_points = hv.Points(markers, ['x', 'y'], columns).opts(color='white', size=4,
                                                                marker='triangle',
                                                                tools=['tap'],
                                                                alpha=alpha)
    return (segments1.opts(color='green', nonselection_alpha=1, alpha=show_segments)
            * segments2.opts(color='green', line_dash='dotted', nonselection_alpha=1, alpha=show_segments)
            * marked_points)

```

Let's take the midpoint of the 2017 times available in the file, as an example time:

```

In [14]: times = zones.BaseDateTime
times = times[times.dt.year == 2017]
midpoint = times.min() + (times.max() - times.min())/2
midpoint

```

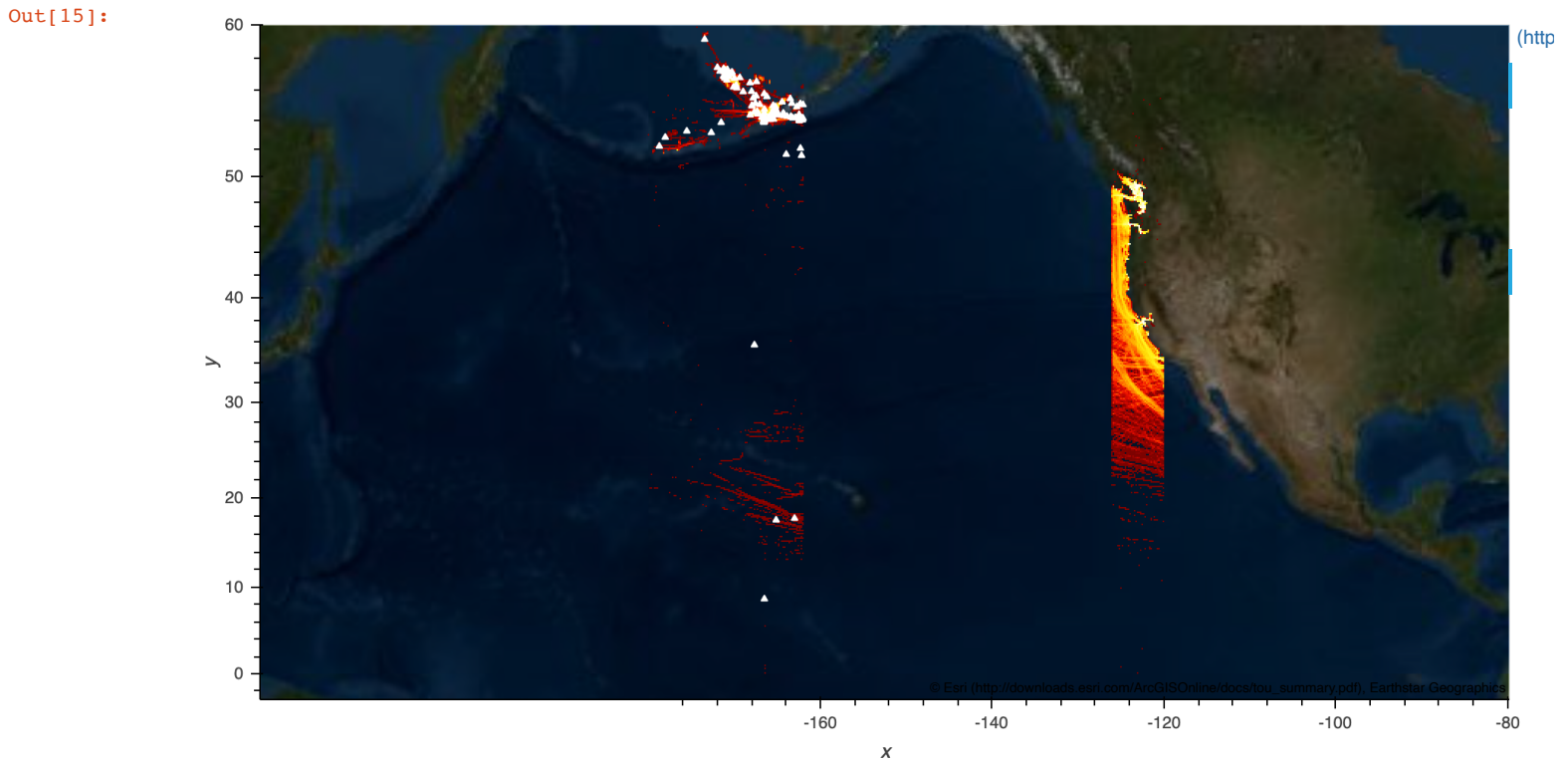
```

Out[14]: Timestamp('2017-01-16 11:59:59')

```

Now we can compute the locations of each vessel near that time, and overlay that on the points and map tiles:

```
In [15]: points = rasterize(hv.Points(zones, ['x', 'y']).redim.range(**bounds)).opts(opts)
        tiles * dynspread(points) * mark_vessels(midpoint).opts(hv.opts.Points(tools=['hover'])))
```



Now when you hover over the white markers indicating the position of a vessel, you should see lots of information about it.

User interface for selecting a given time and vessel

Instead of editing Python code to specify a time, let's add a [Panel \(https://panel.holoviz.org\)](https://panel.holoviz.org) widget to let the user select a specific time interactively:

```
In [16]: dt_input = pn.widgets.DatetimeInput(name='Datetime', value=midpoint)
        dt_input
```

Out[16]:

Datetime

2017-01-16 11:59:59

Also, as the data can be unwieldy in hover form, we can also provide it in a separate table, accessed by clicking on one of the ship markers:

```
In [17]: table_cols = ['MMSI', 'VesselName', 'VesselType', 'Heading', 'CallSign', 'Length', 'Width', 'Cargo']
        empty_df = pd.DataFrame({el:[] for el in table_cols})

        class Drilldown(param.Parameterized):
            selection = param.DataFrame(empty_df)

            @param.depends('selection')
            def update_table(self, *args, **kwargs):
                return pn.widgets.DataFrame(self.selection, show_index=False, width=800)

        drilldown = Drilldown()
```

```
In [18]: def mark_vessel_drilldown(value, index, show_segments=True):
        if len(index) > 0:
            rows = [marked_points.data.iloc[ind] for ind in index]
            df = pd.DataFrame(rows)[table_cols]
            df.columns = table_cols
            drilldown.selection = df

        return mark_vessels(value, show_segments)
```

```
In [19]: points = dynspread(rasterize(hv.Points(zones, ['x','y']).redim.range(**bounds))).opts(opts)

title = "# AIS vessel locations"

message = ("Select a time covered by this dataset and press return to see vessel locations at that time "
          "(after a few seconds, due to unoptimized time-filtering code). Then click on a vessel "
          "to see more information about it. Green lines indicate large AIS gaps, connecting the"
          "vessel to its pings before (solid) and after (dash) the current position.")

dmap = hv.DynamicMap(mark_vessel_drilldown, streams=[dt_input.param.value, hv.streams.Selection1D()])
overlay2 = (tiles * points * dmap)
pn.Column(title, message, dt_input, overlay2, drilldown.update_table).servable()
```

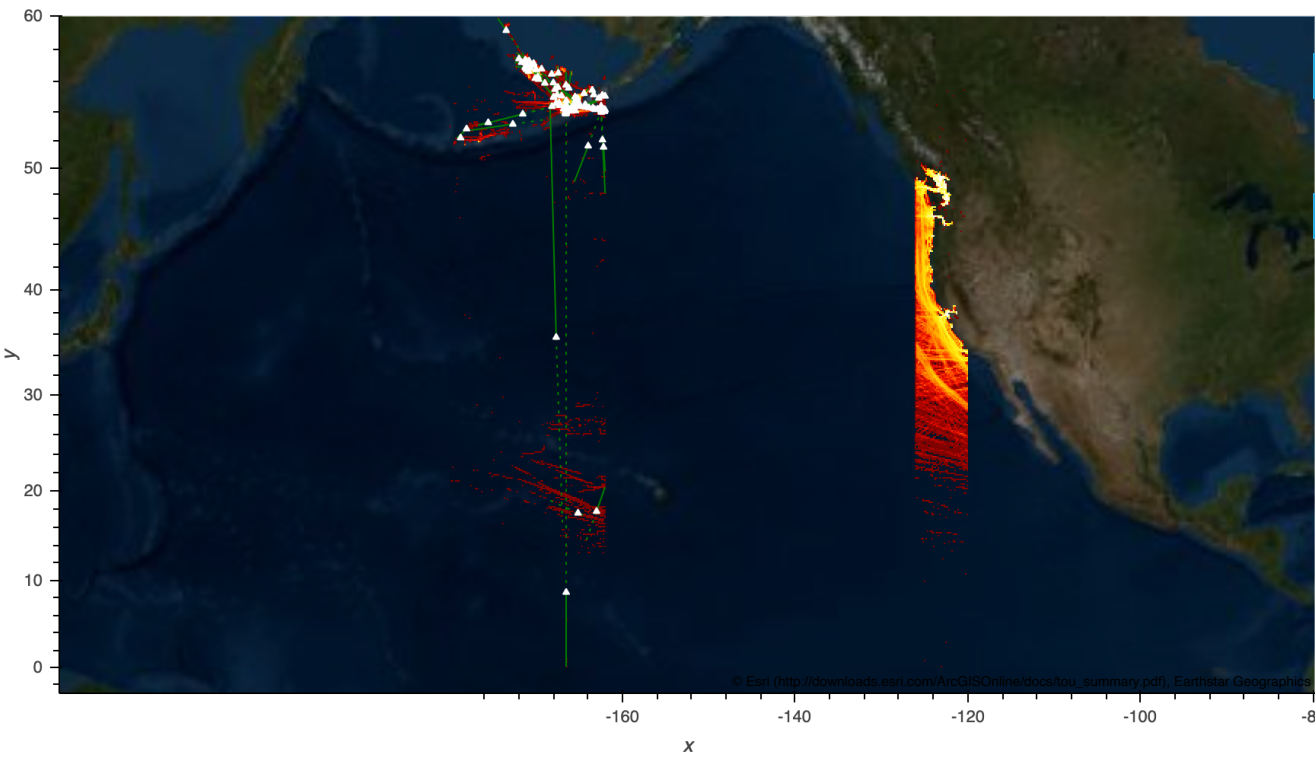
Out[19]:

AIS vessel locations

Select a time covered by this dataset and press return to see vessel locations at that time (after a few seconds, due to unoptimized time-filtering code). Then click on a vessel to see more information about it. Green lines indicate large AIS gaps, connecting the vessel to its pings before (solid) and after (dash) the current position.

Datetime

2017-01-16 11:59:59



MMSI	VesselName	VesselType	Heading	CallSign	Length	Width	Cargo
566,642,000	WAN HAI 512	1,004.0	276.0	9V7578	258.99	37.3	70.0

This app can now be launched as a separate server using a command like `panel serve <notebookname>.ipynb` , then visiting the URL that is printed as output.