Satellite-Vessel Visibility Using High Performance Python

Overview

In this document, we present our approach to solving the satellite-vessel visibility problem using a novel algorithm, written in Python, that is both scalable and of sufficiently high performance that we can drive it from a real-time user interface (also built in Python).

The implementations are available as Python scripts and Jupyter Notebooks. All necessary upstream packages are available in open source, installable with Anaconda's `conda` tool, or `pip` from the PyPI package repository.

The Simple Algorithm

> *Problem Statement:* Given a satellite, determine all vessels and times that it has geodetic overlap. It is acceptable to assume that the field of view of a satellite is ½ of the Earth.

Before describing our optimized solution, we first consider the simple algorithm to the above problem.:

1. Find the appropriate TLE data for a given satellite (index by NORAD ID)
2. Assemble the TLE data so that we can find the most recent/accurate TLE for a given time point.
3. For each vessel, at the time of each AIS ping, compute the satellite's position
4. Compute the "view vector" from the vessel to the satellite

Even using the simplifying assumption that every satellite can see a full half of the Earth, this can be computationally quite expensive. The input dataset consists of nearly

130,000,000 AIS ping points, and although there are only tens of thousands of satellites in the TLE data, there are 120M TLEs to process.

We implemented a naïve version of this "simple algorithm" in `window.py` included in our source, which performs a simple rise-time/set-time calculation of a satellite based on a single TLE, against a single Lat/Long position.

Fast Approach

Based on the requirement for "interactive visualizations" in the Technical Scenario, we create a visibility calculation engine and data architecture that would allow for interactive exploration of different satellites, subsets of vessels (or all vessels), and time points.

Data Processing Pipeline

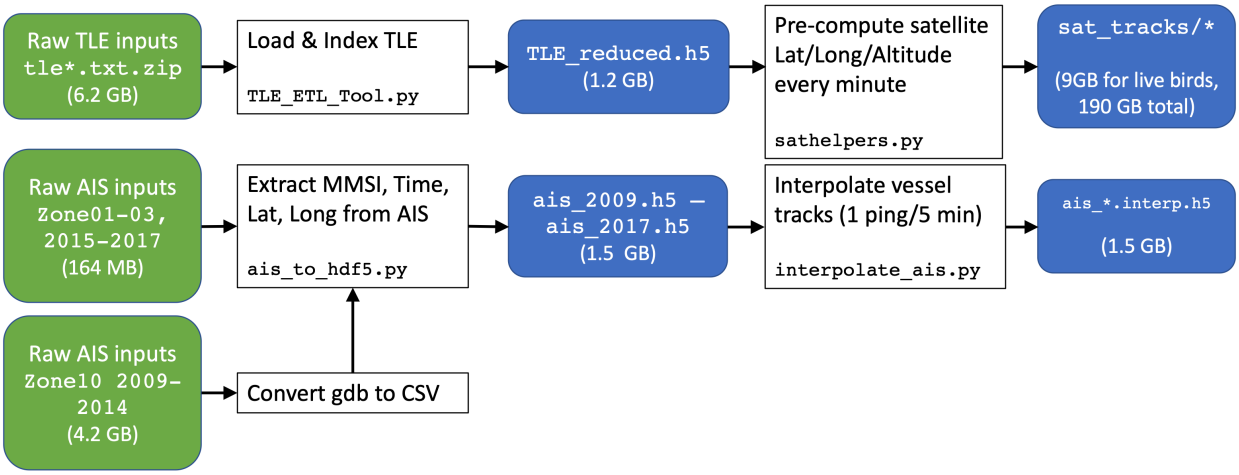
For fast access to large arrays of data, both for computation as well as for visualization, we chose to use the HDF5 format. It is easy and efficient to use from Pandas and NumPy. We wrote several data processing scripts to convert the input raw data from CSV and text formats into HDF5 format. Then we processed them in a variety of ways.

For the TLEs, we stripped out all TLEs that didn't fit the time windows where we had relevant AIS pings. (Based on EDA, we could see that AIS data was only present for the first few months of each year.)

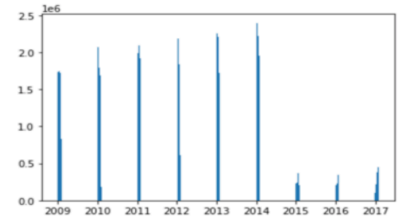
For the AIS data, we:

- Convert Zone10 GDB to CSV format using the ogr2ogr tool from the GDAL open source library:
  `ogr2ogr -f CSV output.csv Zone10_2009_01.gdb -lco GEOMETRY=AS_XYZ`
- Extract only Time, MMSI, Lat, Long from CSVs, and put them into HDF5 files.
- Due to the nature of our algorithm, we also needed to create synthetic interpolated points for vessels that had large gaps in time, but didn't stray very far. (See Algorithm Details section below.)

The following diagram captures the data processing pipeline:



tle2008	4526794	12820	1.0	1968-2009	0.000022	0.0	0.0	1.0
tle2009	5598810	14576	0.0	2006-2010	0.000000	0.0	0.0	0.0
tle2010	5271610	15317	0.0	1971-2011	0.000000	0.0	0.0	0.0
tle2011	5517712	15650	0.0	2010-2012	0.000000	0.0	0.0	0.0
tle2012	5193623	15957	0.0	2010-2013	0.000000	0.0	0.0	0.0
tle2013	2962309	15889	0.0	1962-2014	0.000000	0.0	0.0	0.0
tle2014	3180211	16735	31.0	1962-2015	0.000975	0.0	0.0	31.0
tle2015	3224577	16340	1297369.0	2007-2016	28.690502	0.0	0.0	1297369.0
tle2016	3335414	16426	3271716.0	1968-2017	49.517960	0.0	0.0	3271716.0
tle2017	1949922	17393	5722959.0	2005-2018	74.586834	1.0	1.0	5722957.0
tle2018	3168256	18067	5653035.0	1968-2019	64.083987	28179.0	1.0	5624855.0



AIS Overlap

We saw that the input TLE dataset had approximately 40,785 objects in it.

Given the narrow time windows for the AIS data, we winnowed the TLEs down to roughly 19,400 objects that were relevant in the AIS time windows.

We then further compared the TLE data to the Celestrak SATCAT database, and found we found 3211 objects that are "live".

Of those, only 1033 are "live" during the time windows where we have AIS data.

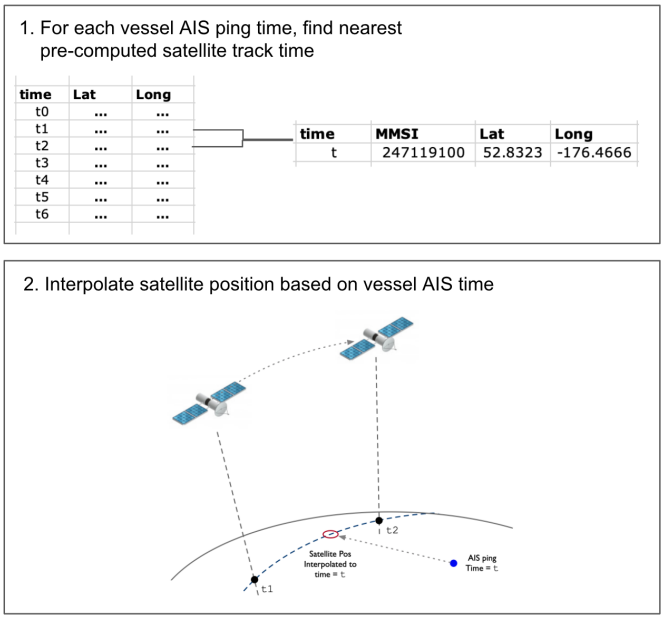
Because the dramatic difference in data size, we constructed our ETL process to separately pre-compute satellite tracks for the "live" objects, during the time windows when we have AIS data. This computation took roughly an hour on a 16-core machine, and yielded a 9.5 GB dataset. We also pre-computed satellite positions for the larger set of 19,400 objects.

Algorithm Details

Our high-performance approach is based on a few insights:

1. Rather than doing a precise calculation of satellite position, we can pre-compute positions at some fixed interval (e.g. 1 minute) and then interpolate its position.
2. Testing the visibility of this satellite track against any individual AIS ping time for any vessel is essentially a parallelizable problem.
3. Furthermore, if we sort the satellite track by time, and we sort all of the AIS pings by time, then we can traverse each list in order and test visibility of each point.

In essentially, we are sorting a discretized and pre-computed satellite track, and then doing a time-based join/intersection against the entire universe of all ship points. The following diagram illustrates the core of the algorithm:



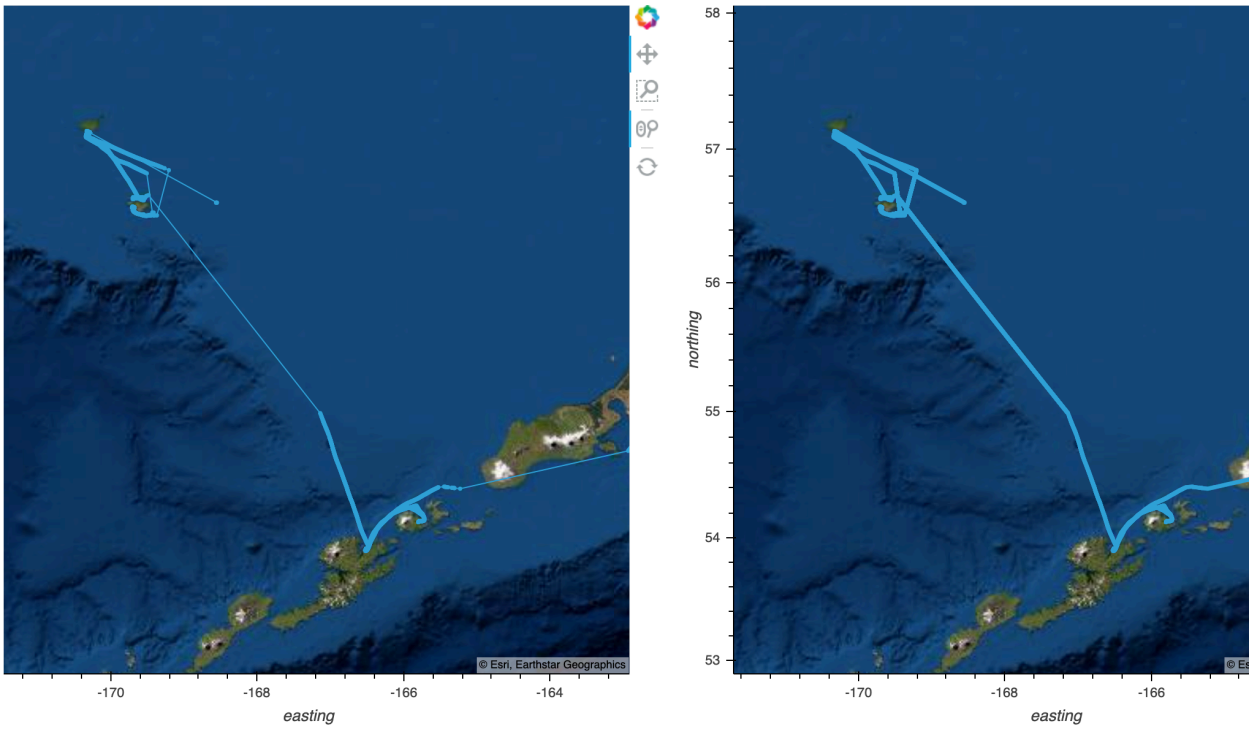
As indicated in this diagram, we use the satellite's altitude to compute a more precise Field of View at each time point. (The code can easily be changed to make the half-earth FOV assumption as well.)

Our intersection calculation is written in Python, but uses the Numba Just-in-Time compiler to accelerate and parallelize the computation with a single decorator. On a 16-core server with 16GB of RAM, our algorithm takes just a few seconds to test visibility of 130,000,000 AIS points.

A Note on Interpolation

Since this is a time-based intersection/join, one issue that arises is that we might not know to test a vessel that has "gone dark" but is essentially stationary or hasn't moved very far from its last seen position. Our simplifying assumption is that if a vessel doesn't stray more than a certain threshold distance between two AIS pings, then we should assume that it traveled a straight line between those pings. As shown in the previous dataflow diagram, the `interpolate_ais.py` script is used to generate interpolated vessel pings for a given input set of pings. The distance threshold and the frequency of interpolated AIS points are configurable. The default distance is 200km, and we generate a synthetic point every 5 minutes.

One example is the vessel with MMSI ID of 43676060. During the 3 days between 2017-01-26 11:47:34 and 2017-01-29 13:34:34, it traveled a distance of only 33km. This can be seen in the image below, on the left, as the thin line connecting its AIS pings around Unalaska and St. George Island to the north.



Our interpolation algorithm fills in this gap with a dense series of points, every 5 minutes, as seen on the right. It can be argued whether this is appropriate in all cases, since a ship can sail a large range in 3 days. For the purposes of this exercise, we felt this was a useful and justified assumption, since any satellite overhead during this time would have a high likelihood of seeing the ship.

The output of the hit-finding intersection algorithm is a list of vessel MMSI IDs, times, and positions where the chosen satellite had geodetic overlap.

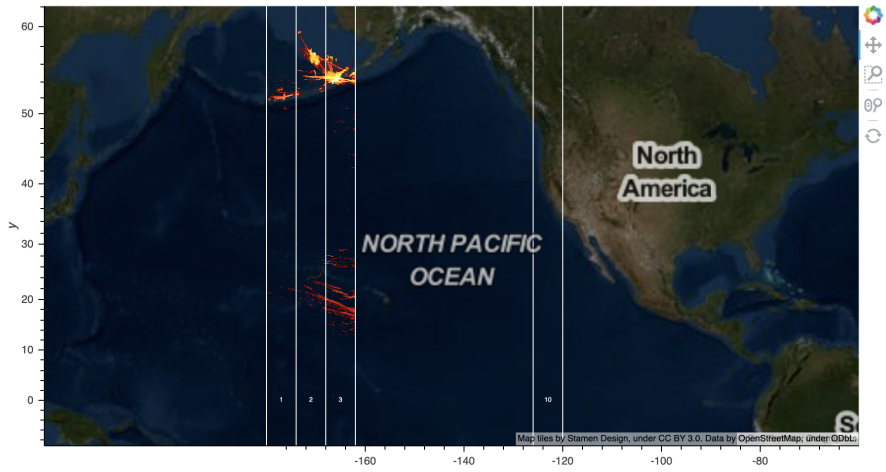
We can also use this compute engine to do a "reverse-search" version of the visibility problem: namely, given a single MMSI ID, find all satellites that were visible during a certain time window. This is implemented in our `reverse-hit-finder.py` script.

Interactive Applications & Data Explorations

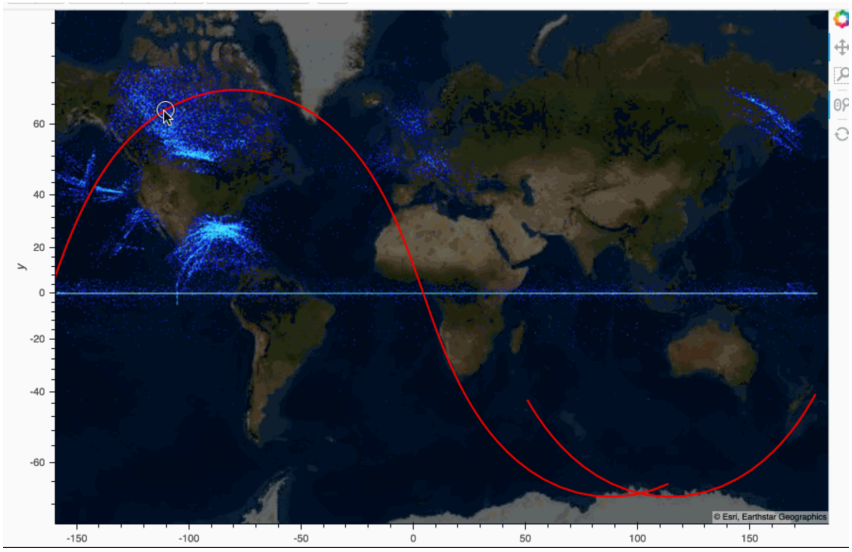
By transforming the data into HDF5, we not only enabled a fast solution of the hit-finding algorithm, but we were also able to power fast, interactive Jupyter Notebooks for data exploration. By using Bokeh, Holoviews, and Datashader libraries, we could easily look at hundreds of millions of points, interactively, in the browser.

The following is an example of a plot created by the `Viewing_AIS.ipynb` notebook. It shows all of the data points for 2017, across zones 1-3.

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



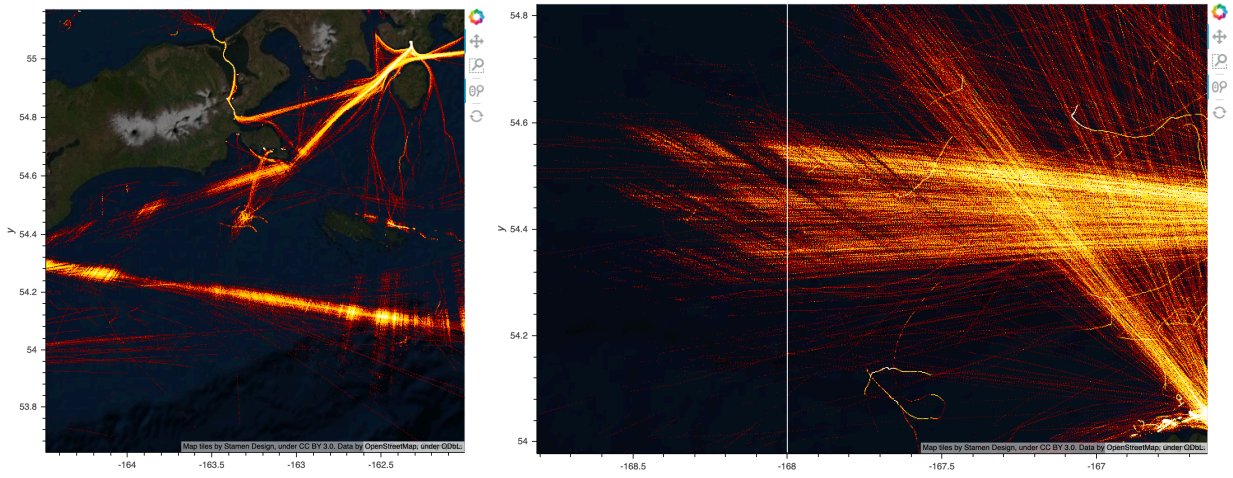
Here are some example visualizations using Holoviews, Bokeh, Datashader to explore vessels in the AIS dataset, and the locations of the TLE data.



Visualizing pre-computed satellite trajectory (red) for a clicked-on TLE point (blue)

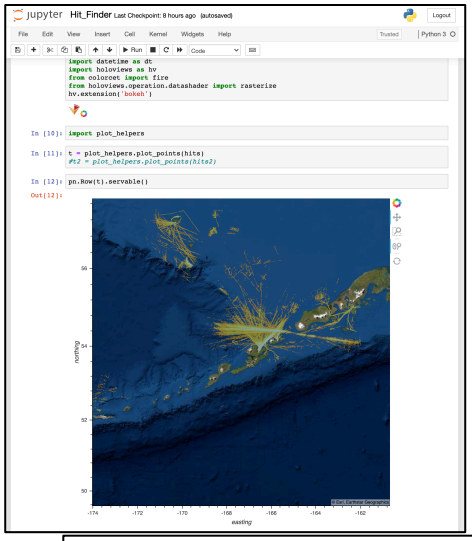
The names of various notebooks & their descriptions can be found at the end of this document, or the README.txt included with the source.

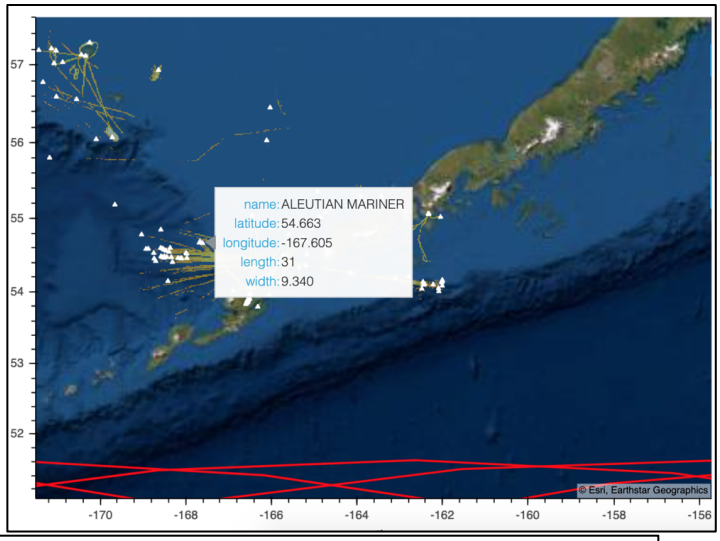
By plotting the large volumes of raw data, we can see structures in the data that would be otherwise hidden or very difficult to extract. For instance, the following plots of the Zone01-03 AIS data shows curious radial banding and "shadows", which may suggest structural blind spots in the AIS receivers.

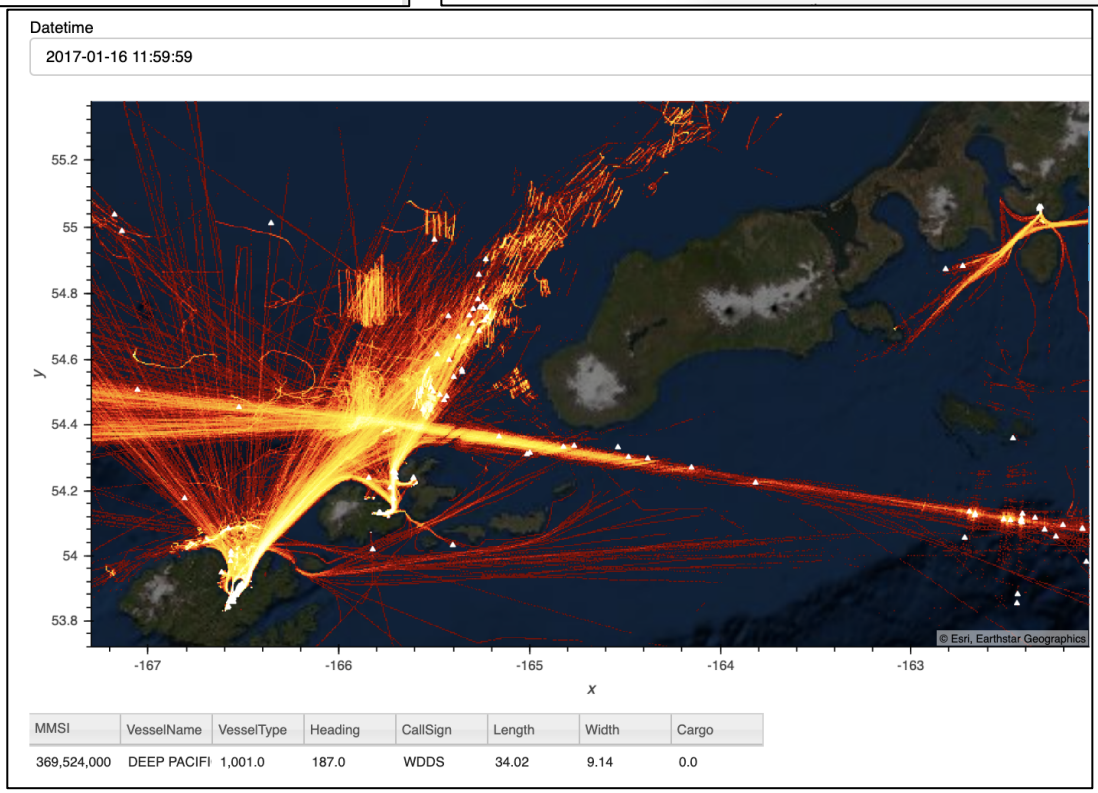


"Hit Finder" Notebook & Dashboard

The main Notebook to run our solution is "`Hit_Finder.ipynb`". (The instructions are detailed in the notebook itself.). There is also a standalone "`Hit_Finder.py`" script which produces an interactive dashboard. Below is a screenshot of the notebook, and the "dashboard" version, showing an interactive plot of an example visibility calculation:







MMSI	VesselName	VesselType	Heading	CallSign	Length	Width	Cargo
369,524,000	DEEP PACIFI	1,001.0	187.0	WDDS	34.02	9.14	0.0

Open Source Tools

We relied on open-source tools for the development of our solution.

Several of these open source tools are primarily developed at Anaconda, Inc. (a member of the AT&T team). They were also partially funded by Air Force Research Laboratory & via DARPA's XDATA program (DARPA-BAA-12-38):

Holoviews, Panel	Rapid development of interactive visualizations and dashboards. Works with Jupyter Notebook as well as standalone web apps.
Bokeh, Datashader	Interactive visualization of millions or billions of points, within the browser or notebooks
Numba	Just-in-time dynamic compiler for numerical Python code. Can provide 10x-100x speedups over plain Python.

For calculation of satellite ephemeris, we used the `skyfield`, `pyephem`, and `pyastro` libraries. (`skyfield` provides a Python-language wrapper for the SGP4 library.) We used `PyTables` to manipulate HDF5 files. In some visualizations, `spatialpandas` was used to enable interactive clicking and selecting of points in a large visualization.

Manifest of Code & Notebooks

`Hit_Finder.ipynb` `hit_finder.py`	The main notebook & dashboard application.
`reverse-hit-finder.py`	Script to find which satellites could see a particular vessel, between some time range.
`ais_to_hdf5.py`	Converts raw AIS CSV files to HDF5, extracting only the MMSI, date_time, Latitude, Longitude fields
`interpolate_ais.py`	Generates interpolated vessel points
`sathelpers.py`	Tool to precompute & propagate a satellite's trajectory
`build_index_parallel.sh`	Script to use GNU Parallel to drive `sathelpers.py` on a multi-core machine
`TLE_ETL_Tool.py`	Converts raw input TLE text files into an HDF5 file, with three nodes: `/tle_raw`, `/tle_reduced`, `/tle_sorted`
`window.py`	Command-line script to perform the "simple algorithm" that computes a single satellite's rise/set times against a single latitude and longitude.
`Viewing_AIS.ipynb`	Notebook to view raw AIS input CSVs. Also shows how to drill down on a simple vessel at a particular time to see its track. (by default, uses 2017 data)
`Viewing_AIS_Categorical.ipynb`	Notebook to view raw AIS data, but colored by vessel type. (by default, uses Zone10 data)
`Viewing_AIS_Paths.ipynb`	Notebook showing tracks of vessels, and the ability to highlight tracks by clicking on the plot
`Viewing_TLEs.ipynb`	Notebook to view & explore raw TLE data