battleVerse Quick-Start Guide

battleVerse is a set of tools designed to help reconstruct test events, and to visualize and analyze sensor system performance
This quick-start guide will help you navigate the battleVerse packages and show how to use the analysis, plotting, and summary functions.
A typical workflow starts with the package **scenarioMaker**, then may continue to either **nautilus** or **sandTable** depending on the user's needs.
For an example workflow, see the last page of this guide.

To install the zipped (tar.gz) packages using R Studio:

1. Save the zipped version of the package build to your hard drive (e.g., "nautilus_1.0.tar.gz")
2. Do not unzip anything
3. In R Studio, go to "Tools > Install Packages…"
4. Choose "Install from: Package Archive File"
5. Choose the zipped version of the package build

To install from the github repository:

1. Make sure you have the devtools package installed: install.packages("devtools")
2. Enter: library(devtools); install_github("battleVerse/scenarioMaker") into the console
3. If you have problems due to a firewall, you may need to install the "httr" package and run the following command: set_config(config(ssl_verifypeer = 0L)) . Check with your IT department first.

You can always find the most recent version of battleVerse at https://github.com/battleVerse.
If you have questions about the battleVerse, contact one of the package authors at:

Dr. Benjamin Ashwell: bashwell@ida.org
Dr. Kevin Kirshenbaum: kkirshen@ida.org

scenarioMaker

The battleVerse runs through **scenarioMaker**.
The function to create a scenario is create_scenario(). A scenario is a list that contains all of the relevant data, and most functions in the battleVerse start with a call to your scenario. Note that this data is still easily accessible: it can be extracted by name from the scenario at any time using the dollar sign (i.e., scenarioName$dataName).

In scenarioMaker, users can reconstruct test events, calculate ranges and bearings to targets, and calculate distance between targets at given times using the function distance_between(). See pages 2-4 for further details. Users can then continue on to nautilus or sandTable, depending on their needs:

For target-track assignments and analysis of sensor system performance

To export to/read from SIMDIS files and reconstruct live fire events

nautilus

nautilus is used to assign radar tracks to targets and calculate sensor system metrics such as error (distance between track and target), track coverage, range and bearing error, detection range, and many more.

See pages 5-10 for further details

sandTable

sandTable contains functions to plot weapon engagements and to import/export SIMDIS formatted data.

See pages 11-12 for further details

Creating Scenarios with scenarioMaker

A scenario is a named list of data frames containing the event data. With few exceptions, all battleVerse functions take a scenario as an input. Start with **scenarioMaker** to create a scenario before continuing with your analysis.

Creating a scenario

```
create_scenario(scenarioName = 'scenario', targetTruth = NA, ownShipTruth = NA, sensorData = NA, engagementData = NA,
  platformInfo = NA, verbose = TRUE, preCalcTargetTrackDist = TRUE)
```

create_scenario() takes as many of the following data frames as possible, but don't worry if you don't have them as you don't need them all to create a scenario. Functions will let you know if they require data that your scenario is missing. To be included in a scenario, these data frames must have the following named variables:

targetTruth (GPS-recorded positions of targets)

a. **lon**: longitude (degrees)
b. **lat**: latitude (degrees)
c. **alt**: altitude (meters)
d. **time**: double (seconds)
e. **heading**: (degrees azimuth)
f. **truthID**: target's ID (factor – string recommended)

ownShipTruth (GPS-recorded position of sensor system)

a-f: Same as Truth Data

sensorData (target positions/tracks as recorded by sensor system)

a-e: Same as Truth Data
f. trackNum: track number (factor – int recommended)

engagementData (who fired at whom with what, used by sandTable)

a. **time**: time of engagement (double, seconds)
b. **source**: name of platform doing the shooting
c. **target**: name of target
d. **weapon**: name of weapon
e. **kill**: 0 = no kill; 1 = kill
f. **color**: color of line to be drawn – must be valid SIMDIS choice

platformInfo (target details, used by sandTable and SIMDIS)

a. **truthID**: target's ID
b. **platformIcon**: name of platform icon – SIMDIS format
c. **platformType**: type of platform – SIMDIS format
d. **trackColor**: color of the track – SIMDIS format

If your data is **not in lat/lon/alt format**, we have transformation functions to get you there:

If your data is in offset format (meters North/East/Up from sensor):

```
transform_offset_to_latlon(referenceData, relativeSensorData)
```

If your data is in range/bearing to target format (uses altitude, not elevation):

```
transform_bearing_range_to_latlon(referenceData,
  relativeSensorData)
```

The **output** of create_scenario() contains **all of the input data frames** as well as **some new ones**:

targetOwnShipDistance – ranges, bearings, target aspects, and errors from the sensor system to every truth position of all targets. This is calculated only if you input both targetTruth and ownShipTruth.

targetTrackDistance – ranges, bearings, target aspects, and errors associated with every possible target-track pair. This is calculated only if you input both targetTruth and sensorData.

trackOwnShipDistance – ranges, bearings, target aspects, and from the sensor system to every sensor-determined position of all targets. This is calculated only if you input both sensorData and ownShipTruth.

Exploration Figures

We have created functions to help users explore the scenario data. Many of these functions have both ggplot and plotly versions of the figure: the ggplot versions are easy to modify and add layers, the plotly versions are interactive and may help users to inspect their data.

Here we show an example scenario included in scenarioMaker, example2_scenario. This scenario was created with the call:

```
example2_scenario <- create_scenario(scenarioName = "scenario2", targetTruth = example2_truthData,
                          ownShipTruth = example2_ownShipData, sensorData = example2_sensorData)
```

plot_sensor_and_truth_data(scenario, offset, textSize, hideLegend, useDefaultColors)

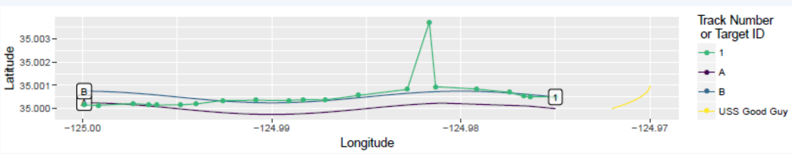


Figure showing overall data. Target labels are at first point in time, track labels are at last point.

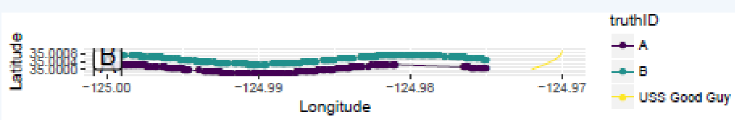
plot_sensor_and_truth_data_plotly(scenario, useDefaultColors)

Interactive version of plot_sensor_and_truth_data()

plot_sensor_and_truth_data_with_altitude(scenario, offset, textSize)

Same as plot_sensor_and_truth_data() but intended for scenarios with nonzero altitudes

plot_truth_data(scenario, offset, textSize, hideLegend, useDefaultColors)



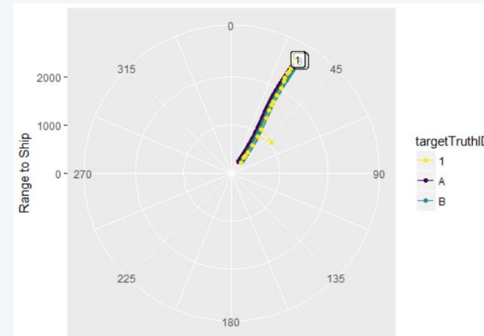
Same as plot_sensor_and_truth_data() but only showing truth data

plot_truth_data_plotly(scenario, offset, textSize, useDefaultColors)

Interactive version of plot_truth_data()

plot_relative_truth_and_sensor(scenario, offset, textSize, hideLegend, useDefaultColors, plotWhich)

plot_relative_truth_and_sensor_plotly(scenario, offset, textSize, plotWhich)



Polar plot showing range and bearing to sensor system

plot_distance_data(scenario, hideLegend, useDefaultColors)

plot_distance_data_plotly(scenario, useDefaultColors)

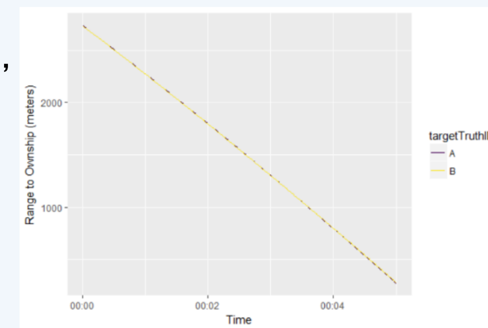


Figure showing target range to sensor system vs. time

plot_truth_gaps(scenario)

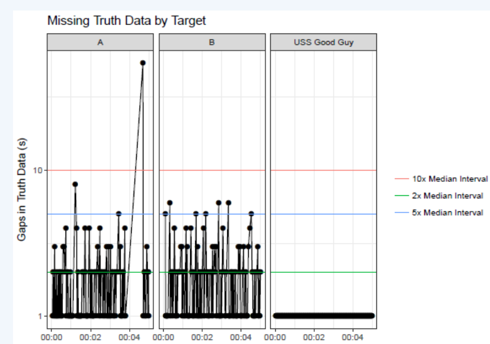


Figure showing gaps in truth data. Use caution as battleVerse interpolates as usual through these regions

Useful additional function: distance_between()

After create_scenario(), the most useful function in scenarioMaker might be **distance_between()**. create_scenario() automatically calculates the distance of all targets and tracks to the sensor system (called 'ownShip' within functions). During scenario reconstructions, however, it may be useful to know the distance and relative bearing between other targets. For example, if target A fired at target B at times t_1, t_2, and t_3, an analyst might want to know how far apart those targets were at that time. distance_between() calculates the distance and bearing between any specified targets.

As an example, let's look at example2_scenario from before. There are two targets, A and B. We already have the range, relative bearing, and target aspect from the perspective of own ship and treating A and B as targets. If, however, A fired at B at times t = 15, 35, and 65, we might want to know how far away B and own ship were from target A. To do that, we would use distance_between() :

```
distanceExample1 <- distance_between(scenario = example2_scenario, timeList = c(15, 35, 65), ownShipList = c("A"),
targetList = c("B"), windowScale = 20)
```

This treats anything names in ownShipList as the origin (here, "A"), and calculates distances to all targets (here, "B") at the specified times. The output is the following useful data frame:

time	ownShipTruthID	targetTruthID	ownShipLon	ownShipLat	ownShipAlt	slantRange	groundRange	targetAspect	trueBearingToTarget	relBearingToTarget	targetLon	targetLat	targetAlt	targetHeading
15	A	B	-124.9988	35.00023	0.000000e+00	55.47027	55.47027	87.91231	4.019892e-09	267.9084	-124.9987	35.00073	0.000000e+00	92.08769
35	A	B	-124.9971	35.00015	0.000000e+00	55.47029	55.47029	85.67158	3.600000e+02	265.6701	-124.9971	35.00065	9.313226e-10	94.32842
65	A	B	-124.9946	34.99997	9.313226e-10	55.47029	55.47029	84.59474	0.000000e+00	264.5948	-124.9946	35.00047	1.862645e-09	95.40526

Note that a list of "own ships" and "targets" can be specified. create_scenario() automatically calculated distances from ownShip to A and ownShip to B. If we wanted to get distances from A and B to A, B, and ownShip, we could write:

```
distanceExample2 <- distance_between(scenario = example2_scenario, timeList = c(15, 35, 65), ownShipList = c("A", "B"),
targetList = c("A", "B", "USS Good Guy"), windowScale = 20)
```

Which then gives:

time	ownShipTruthID	targetTruthID	ownShipLon	ownShipLat	ownShipAlt	slantRange	groundRange	targetAspect	trueBearingToTarget	relBearingToTarget	targetLon	targetLat	targetAlt	targetHeading
15	A	USS Good Guy	-124.9988	35.00023	0.000000e+00	2625.27175	2625.27177	29.55184	8.825054e+01	356.15893	-124.9700	35.00095	-1.862645e-09	238.71519
15	A	B	-124.9988	35.00023	0.000000e+00	55.47027	55.47027	87.91231	4.019892e-09	267.90839	-124.9987	35.00073	0.000000e+00	92.08769
35	A	USS Good Guy	-124.9971	35.00015	0.000000e+00	2471.21884	2471.21886	29.41158	8.811127e+01	353.78139	-124.9700	35.00088	-9.313226e-10	238.71521
35	A	B	-124.9971	35.00015	0.000000e+00	55.47029	55.47029	85.67158	3.600000e+02	265.67012	-124.9971	35.00065	9.313226e-10	94.32842
65	A	USS Good Guy	-124.9946	34.99997	9.313226e-10	2237.41833	2237.41834	28.97302	8.767422e+01	352.26899	-124.9701	35.00078	0.000000e+00	238.71524
65	A	B	-124.9946	34.99997	9.313226e-10	55.47029	55.47029	84.59474	0.000000e+00	264.59477	-124.9946	35.00047	1.862645e-09	95.40526
15	B	USS Good Guy	-124.9987	35.00073	0.000000e+00	2624.16408	2624.16410	30.76250	8.946120e+01	357.37351	-124.9700	35.00095	-1.862645e-09	238.71519
15	B	A	-124.9987	35.00073	0.000000e+00	55.47027	55.47027	267.90839	1.800000e+02	87.91231	-124.9988	35.00023	0.000000e+00	92.09161
35	B	USS Good Guy	-124.9971	35.00065	9.313226e-10	2470.01288	2470.01290	30.69771	8.939740e+01	355.06898	-124.9700	35.00088	-9.313226e-10	238.71521
35	B	A	-124.9971	35.00065	9.313226e-10	55.47029	55.47029	265.67012	1.800000e+02	85.67158	-124.9971	35.00015	0.000000e+00	94.32988
65	B	USS Good Guy	-124.9946	35.00047	1.862645e-09	2235.85433	2235.85434	30.39347	8.909467e+01	353.68940	-124.9701	35.00078	0.000000e+00	238.71524
65	B	A	-124.9946	35.00047	1.862645e-09	55.47029	55.47029	264.59477	1.800000e+02	84.59474	-124.9946	34.99997	9.313226e-10	95.40523

As a final note, these can be saved to your scenario to save them and avoid recalculating:

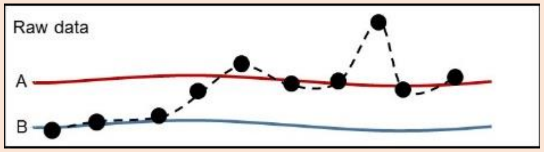
```
example2_scenario$customDistance <- distanceExample2
```

Target assignment with nautilus

We created **nautilus** to automate track assignment, at is, assigning a sensor system track to a target and calculating errors. Because some systems produce a large number of false tracks, nautilus will also remove false tracks that are outside a user-specified cutoff distance. Below we show the four target assignment methods we created and how to specify cutoff distance.

Nautilus takes each sensor point, interpolates all of the truth data targets to the time of that sensor point, then assigns the sensor point to the nearest target using one of four methods. **target_assignment()** returns the scenario with the assignment parameters. The call to this looks like:

```
scenario <- target_assignment(scenario, method, cutoff, ...)
```

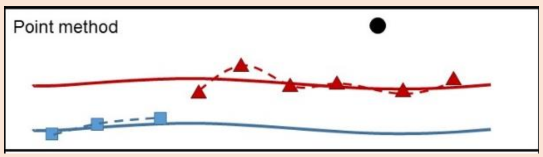


Here is an example of one radar track on two targets. There are two big features to look out for here that show up in a lot of data: track switching targets and inaccurate points in a track. The four methods may return slightly different results.

Note: window methods require additional argument, **windowSize.** Very small windowSize is equivalent to point method. Very large windowSize is equivalent to whole track method

```
method = "point"
```

Point method evaluates each point in a track separately

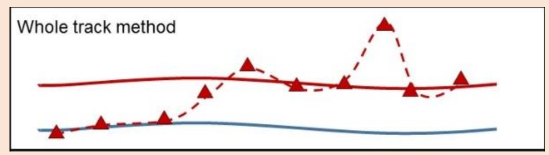


Advantages: Good, quick first pass. Easy to explain. Handles track switching well

Disadvantages: May bias toward high accuracy by excluding more valid points

```
method = "wholeTrack"
```

Whole track method evaluates each track, picks closest average target

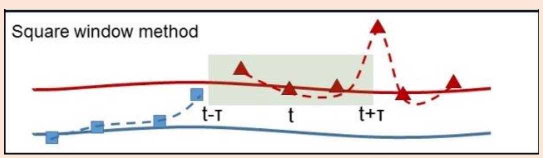


Advantages: Better approximation of what operator is seeing

Disadvantages: Bad determination of track switching, track coverage

```
method = "windowSquare", windowSize = 2τ
```

Square window method averages surrounding points out to time $t \pm \tau$

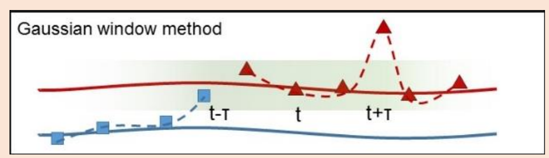


Advantages: Good estimate of track continuity, handles track switching

Disadvantages: Difficult to explain, changing window can dramatically change results

```
method = "windowGauss", windowSize = 2τ
```

Gaussian window method averages all points in a track weighted by st.dev = τ



Advantages: Good estimate of track continuity, handles track switching

Disadvantages: Difficult to explain, changing window can dramatically change results

Cutoff options

3 choices to set cutoff: 1, 2, or 3 numbers

Cutoff	Units:
`cutoff = locationError`	[m]
`cutoff = c(xyError, altError)`	[m], [m]
`cutoff = c(rangeError, bearingError, altError)`	[m], [deg], [m]

NOTE: **we strongly recommend using only locationError cutoffs**

Points outside cutoff are flagged as false tracks. This is necessary for analysis when there are clutter or false track points. Failure to remove these will unfairly penalize system accuracy.

Caution: Cutoff is the maximum error. If you set locationError = 200 m, the largest error you can calculate is 200 m. This may be realistic or incorrect, you must use your best judgement in setting cutoff.

Advanced Topic - Comparison of Track Assignment Methods

Nautilus uses one of four methods (point, Gaussian window, square window, and whole track) to determine which sensor points are false tracks, and of the remainder, which sensor point should be assigned to which truth track. Point, in which each sensor point is assigned to a target with no reference to other sensor points, and whole track, in which entire tracks are assigned to targets, are opposites in how they treat the data. The two window methods are intermediate, with very small window sizes acting like the point method, and very large window sizes acting like the whole track method. There are several important things to remember when using Nautilus:

1) **There is no overall best track assignment method**. Each has its own advantages and disadvantages.
2) Your choice of cutoff (and window size, if applicable) is just as important as your choice of track assignment method.
3) In some situations, track-to-truth associations are **fundamentally ambiguous**, and there may be **no right answer.**
4) What matters is that the track assignments are **reasonable**, and that calculated results of interest (e.g., bearing error), are **not strongly sensitive** to minor changes in cutoff and window size.
5) **Always look at your track assignments** (plot_target_assignments and plot_target_assignments_plotly) **before using the results.**

Point Method

General Behavior: Considers each sensor point individually. Because it marks all points outside the cutoff as false tracks, and always associates points with the closest target, it tends to minimize overall position error. By the same token, because it always assigns points to the closest target, it will aggressively switch between targets to minimize position error, and will aggressively exclude sensor points outside the cutoff, even when they are part of otherwise accurate sensor tracks.

When to use it:
- When evaluating sensor accuracy (i.e., was there a target there when the sensor said there was)
- When evaluating very large data sets

When not to use it:
- When evaluating track continuity (how long tracks remain unbroken on the same target)

Window Methods

General Behavior: Uses neighboring sensor points in the same track to determine whether a sensor point is a false track, and which target it should be assigned to. The window methods tend to smooth out minor breaks and discontinuities in data. As a result, it can produce more reasonable answers, particularly for noisy data where tracks wander between targets. However, over-smoothing the data can result in unexpected target assignments and false tracks, and can provide an unrealistic assessment of performance.

When to use it:
- When evaluating noisy data, especially data with closely spaced targets and tracks wandering between them
- When evaluating how well the sensor supported overall situational awareness

When not to use it:
- When evaluating very large data sets

Whole Track Method

General Behavior: Considers each track as a unit, and either marks the entire track as a false track, or assigns the entire track to a single target. This method does not allow a track to change targets. While this produces very clean results, it also has the highest potential of producing unreasonable results.

When to use it:
- When evaluating clean data with little or no movement of tracks between targets
- When suppressing clutter/false tracks as part of a 2-pass assignment

When not to use it:
- When evaluating results in which tracks move between targets
- When evaluating track coverage (how long each target is tracked)

Summary Figures

If you have already called target_assignment(), the output scenario has a data frame with two new items:

assignmentData: the data frame containing the position error, range error, bearing error, etc. derived from the difference between track and target

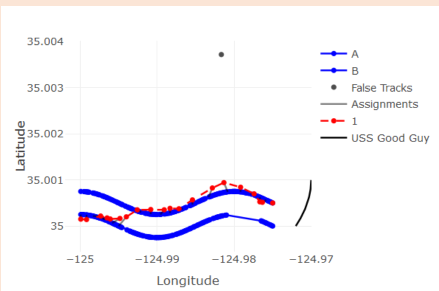
assignmentParameters: A list of the parameters input by the user into target_assignment()

These plot functions show summaries of the results stored in assignmentData. Note that if you have need of a figure that we have not already created, you can use the call scenario$assignmentData to extract your data and create your own figures.

```
plot_target_assignments(scenario,
  scalePoints, textSize, showFalseTracks,
  hideLegend)
```

Figure showing which tracks were assigned to which targets. Target labels are at first point in time, track labels are at last point.

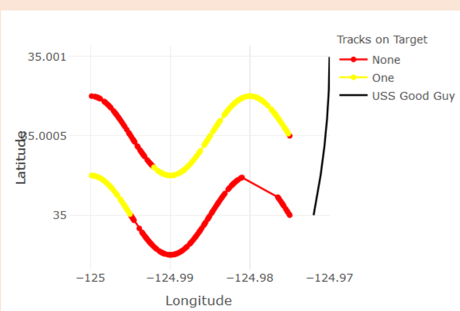
```
plot_target_assignments_plotly(scenario)
```



Interactive version of plot_target_assignments()

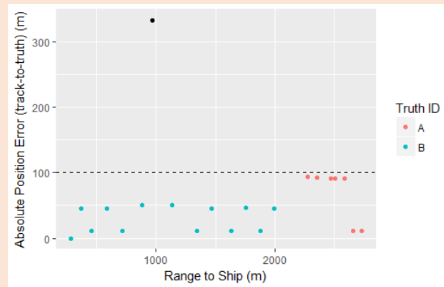
```
plot_track_status(scenario, scalePoints,
  textSize, showFalseTracks, hideLegend)
```

```
plot_track_status_plotly(scenario)
```



Interactive figure showing whether a target had 0, 1, or multiple tracks assigned to it

```
plot_error(scenario, rangeCutoff, xTerm,
  yTerm, colorTerm, doFacet,
  plotFalseTracks)
```

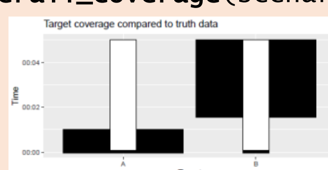


Versatile figure showing [location error, bearing error, or downrange error] by [time or range to ship] and split by [target or track]

```
plot_scatterplot_with_density(scenario
  , xValue, yValue)
```

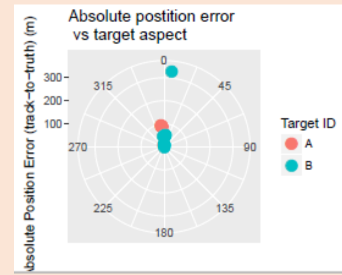
Scatterplot showing systematic bias in bearing and downrange error

```
plot_overall_coverage(scenario)
```



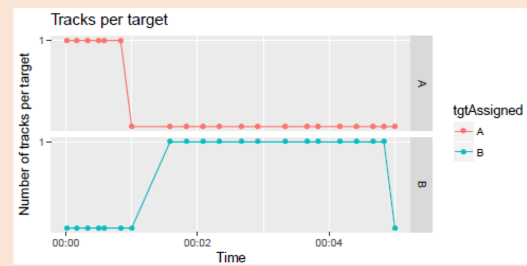
Overall track coverage by target. Can be compared to truth data for each target

```
plot_polar_error(scenario, angleTerm,
  rTerm, colorTerm, doFacet,
  plotFalseTracks)
```



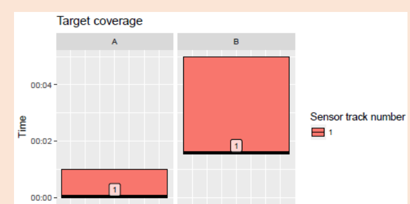
Polar plot showing [location error, bearing error, or downrange error] by [bearing or target aspect] and split by [target or track]

```
plot_tracks_per_target(scenario)
```



Tracks per target by time

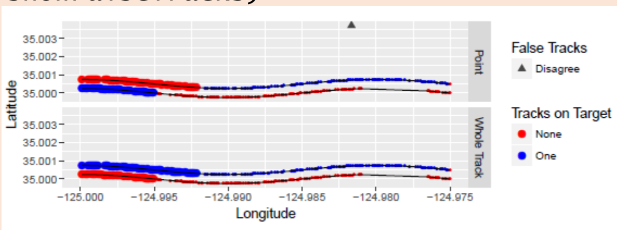
```
plot_coverage_by_target(scenario,
  labelOffset, textSize)
```



Track coverage by target and track number

Summary figures, cont'd

plot_compare_track_status(assignmentDataList, nameList, truthData, showFalseTracks)



Comparison of two plot_track_status() figures. Used to investigate difference in target assignments between methods

Summary tables

summarize_performance(scenario)

Table with output: detection range per target, average position error, average bearing error, average downrange error

summarize_time_tracked(scenario)

Table with output: Total time tracked per target, number of unique tracks following each target, first and last time each target was tracked

summarize_time_exists(scenario)

Table with output: total time that each target existed based on truth data

summarize_time_exists() and summarize_time_tracked() can be combined to give the fraction of time that each target is tracked:

```
# A tibble: 2 x 4
  targetID `Total Time Tracked` timeExists `Fraction of time tracked`
    <fctr>               <time>     <time>                      <dbl>
1        A              59 secs   299 secs                  0.1973244
2        B             205 secs   299 secs                  0.6856187
```

Advanced target assignment: two-pass method

Some sensor systems produce a lot of false tracks that need to be removed before analysis. With faster methods like the point or whole track methods this is not a problem, however with the slower window methods this can be a long computation in scenarios with tens of thousands of data points. Analysts might want to use a faster method first to cut out as many false tracks as possible, and then use a slower method for a more accurate track assignment. We have included a way to do this that we call the **two-pass method**.

In the two-pass method, target_assignment() is run twice; the assignments are dropped after the first pass, but the points flagged as false tracks are kept. Then, during the second pass, those false track points are ignored during target assignment. This speeds up the computation and allows analysts to better cut off false track data.

We have had success with a whole track method as the first pass, which will cut out all whole tracks that are false tracks, but not throw out any points that the sensor system believed belonged to one track. For the second pass, analysts may choose whichever method they prefer.

For example, a call to this with a whole-track method first followed by a window method might look like:

```
myScenario <- example2_scenario %>%
              target_assignment(method = 'wholeTrack', cutoff = 100) %>%
              target_assignment_secondpass(method = 'windowSquare',
                                           cutoff = 300, windowSize = 20)
```

There is an option to completely remove the false tracks flagged during the first pass. This is useful when you have an extreme number of false tracks, many of which are obviously false tracks. These will be entirely excluded from the final data frame to make plotting and analysis faster.

Note: example2_scenario does not have any entire tracks that are false tracks, however example1_scenario does.

Interactive Nautilus

Interactive Nautilus is a GUI to facilitate using Nautilus for rapid analysis or for users with less experience using the R scripting language.
To use Interactive Nautilus:

1. Load and format data before launching Interactive Nautilus (see pgs. 1 and 2 of quick start guide)

2a. Go to “AddIns > Interactive Nautilus” or;
2b. Enter in the console:
```
nautilus:::nautilus_interface()
```

3. Continue your analysis by setting parameters to assign tracks to targets and graphing results

Parameters Tab

When in doubt, **use the tutorial**

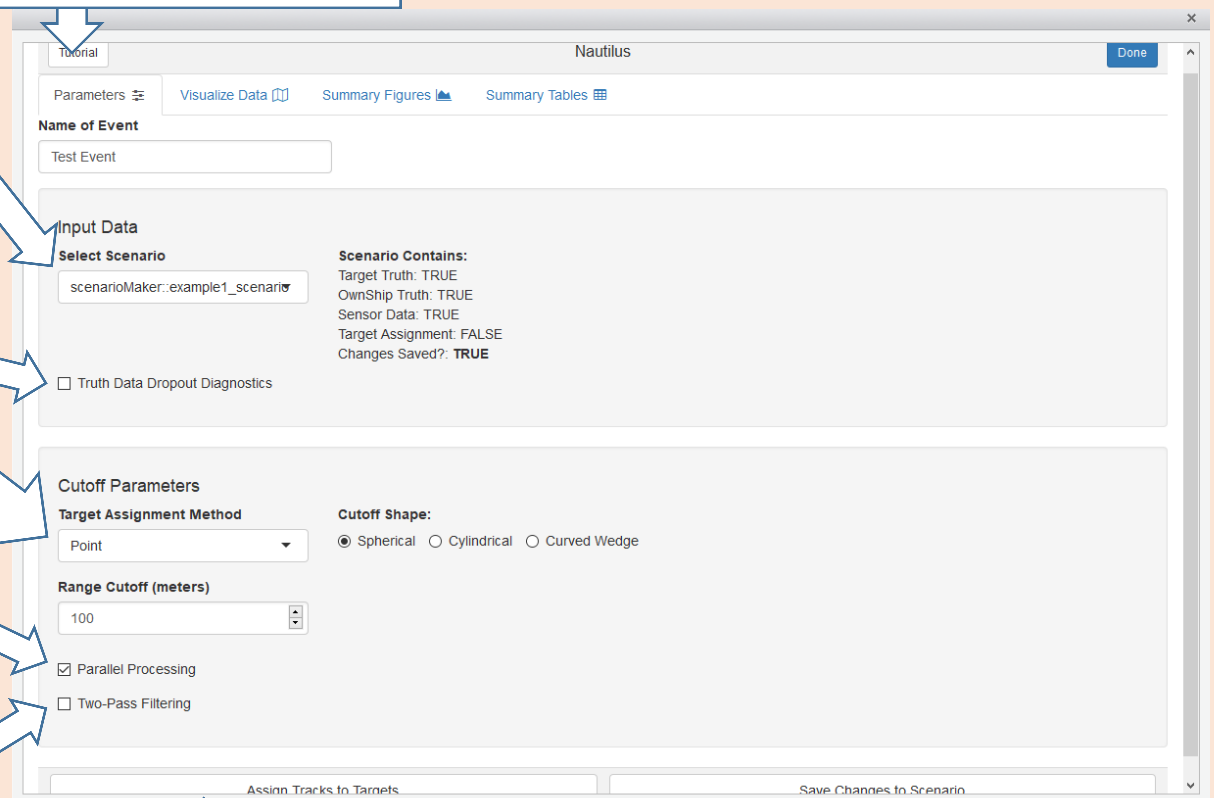
Choose which scenario in your workspace you want to analyze. If you don’t have any scenarios it will automatically load example 1 for you to experiment with

Inspect the truth data for dropouts

Choose target assignment method and set cutoff parameters (determining false tracks).

Parallel processing may speed up analysis times for large data sets when using window methods

Two-pass filtering may speed up analysis times when using window methods for data for data sets with a large number of false track points



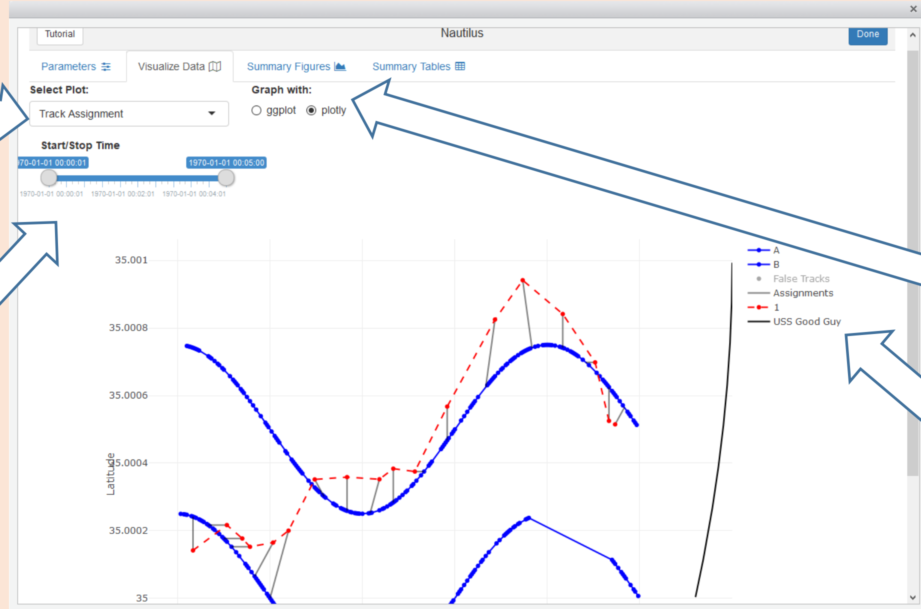
Click “Assign Tracks to Targets” when you’re done setting the assignment parameters, then move on to the Visualize Data tab. Large data sets may take a few moments to analyze

Once you’re done creating figures and tables, come back to this tab to save your target-track assignments, either to the R environment or to your computer

Visualize Data Tab

Choose which plot you want to inspect. Some can be used during your initial data inspection phase while others require you to have clicked “Assign Tracks to Targets” already

Use the slider to view a limited time window



Can use ggplot to make figures or plotly to make interactive ones

In plotly you can turn tracks on and off and zoom in to specific areas

Summary Figures Tab

Select figure

Change axes

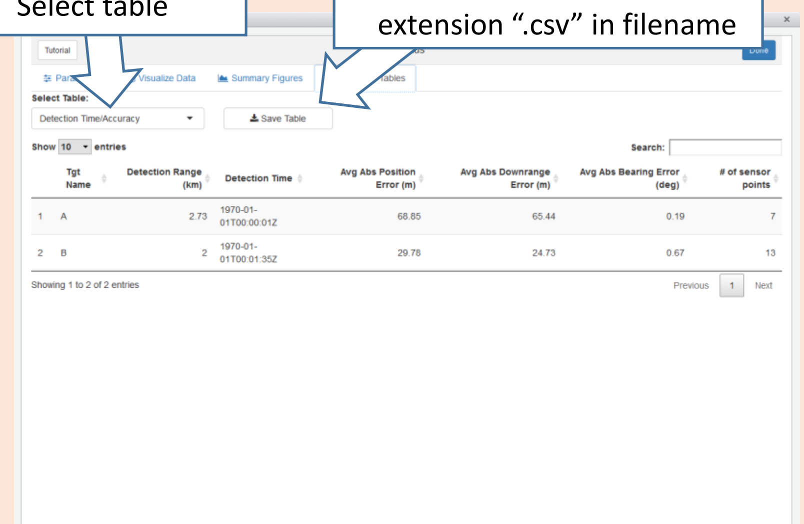
Change grouping parameter

Facet and/or plot the points determined to be false tracks

Summary Tables Tab

Select table

Saves table as .csv . Include extension “.csv” in filename



sandTable

We created sandTable to help with engagement reconstruction

Import/export SIMDIS data

read_ASI`(ASIFilePath, timeFormat, timeZone)`

Reads data from SIMDIS files. timeFormat is set by default, however this can be changed if your time is in a different format

export_scenario_to_ASI`(scenario, outFileName, drawEngagements, drawSensorTracks, persistPosition, prefFile)`

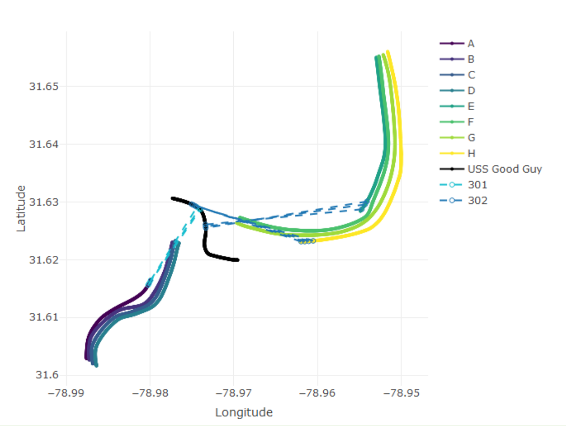
Writes data to SIMDIS format

Additional engagement plots

Note: these figures are using example1_scenario that is included in scenarioMaker

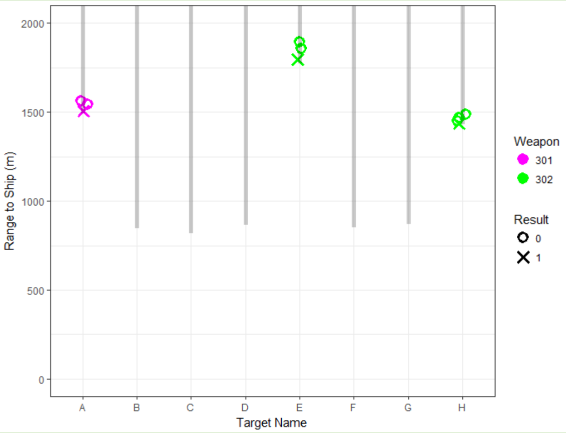
plot_add_engagements`(scenario, useDefaultColors)`

ggplot figure showing the truth positions of ownship and targets as well as all weapon engagements



plot_money_chart`(scenario)`

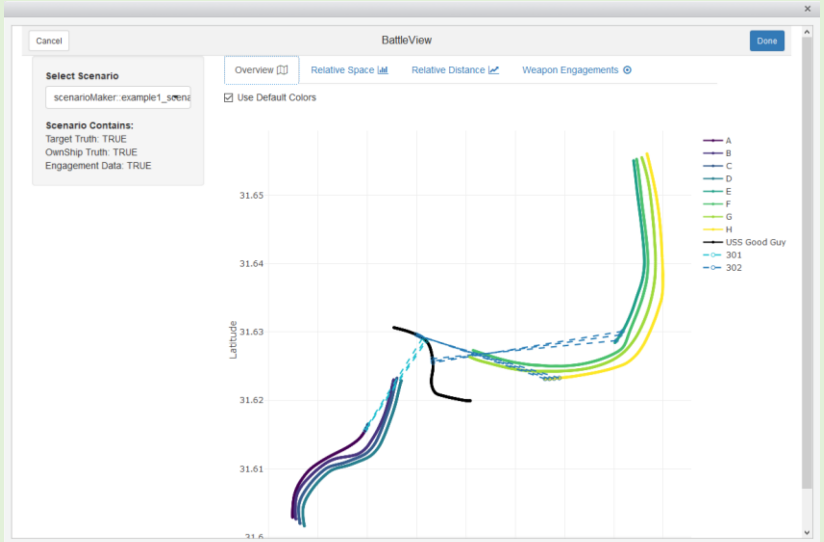
ggplot figure showing the range of each target to ownship, the range of each engagement, and whether that engagement destroyed the target



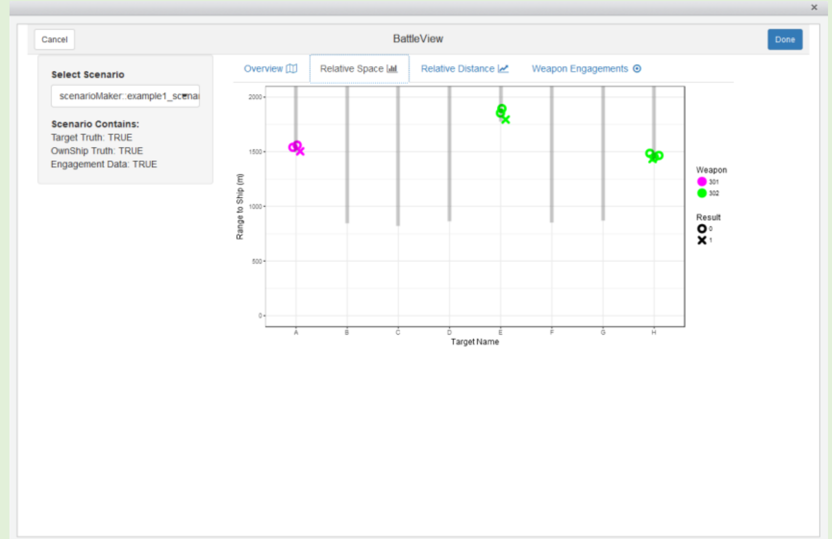
sandTable GUI for interactive reconstructions

Run the GUI by either going to Addins > sandTable or running sandTable::sandTable_interface() from the console. The GUI currently has 4 tabs:

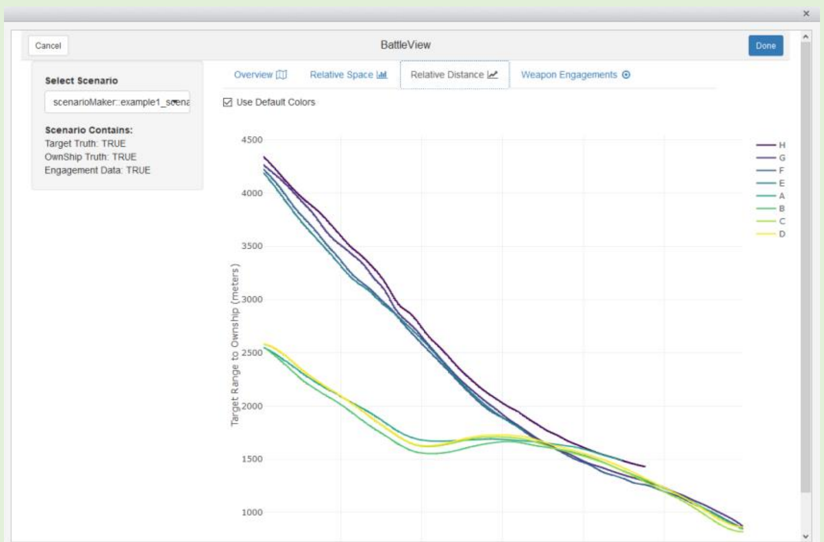
Overview: a display of plot_add_engagements(). Shows truth positions of targets and ownship as well as engagements



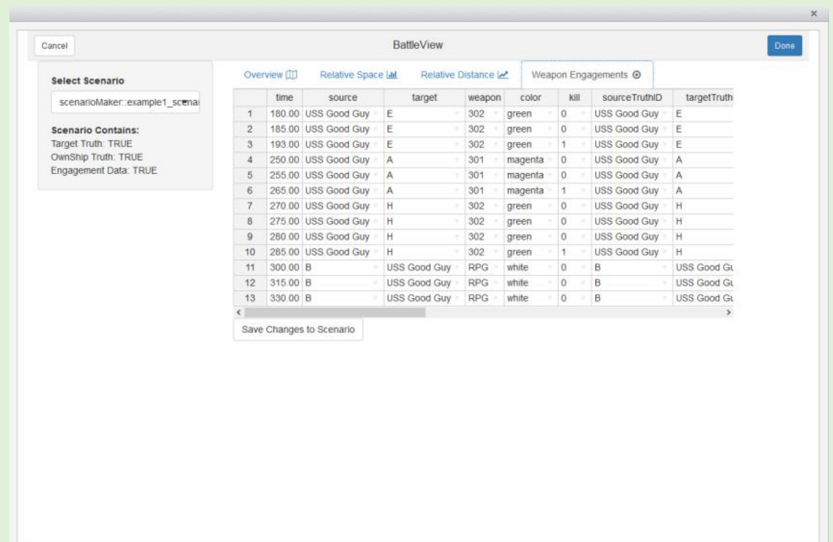
Relative Space: a display of plot_money_chart(). Shows target distance to ownship, distance of engagements, and kill/no kill



Relative Distance: a display of plot_distance_data(). Shows range to ownship vs. time



Weapon Engagements: Editable table of weapon engagements. Changes can be saved to scenario



Example workflow

Let's look at how we might begin to analyze example 1 in scenarioMaker (that is, the example data files included with the package starting with "example1_...". This example workflow represents only part of how the battleVerse can help you with your analysis. See the documentation and tutorials for more help, or email one of us:

Dr. Benjamin Ashwell: bashwell@ida.org
Dr. Kevin Kirshenbaum: kkirshen@ida.org

Begin by loading the battleVerse libraries:

```
library(scenarioMaker); library(nautilus); library(sandTable)
```

1. Check data format

```
names(example1_truthData)
> "lon" "lat" "truthID" "time" "alt" "heading"
```

Check that your data has the correct columns and column names. Our example data is already formatted correctly. If your columns are named something other than what is shown in page 2 of this guide then create_scenario() will warn you.
If your data is not in lat/lon/alt format, we have create some functions to help you convert (see functions starting with "transform_...").

2. Create scenario

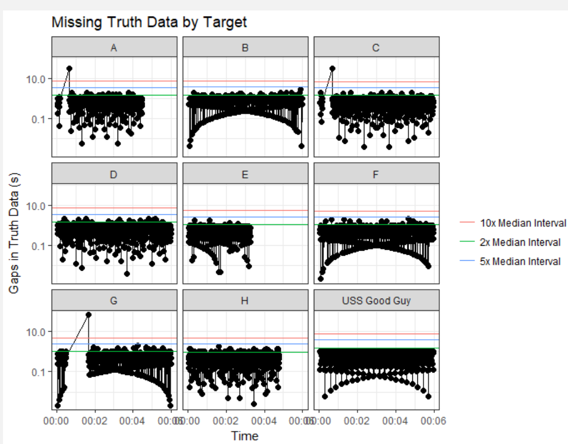
```
myScenario <- create_scenario(scenarioName = "Example
   Scenario", targetTruth = example1_truthData_dropout,
   ownShipTruth = example1_ownShipData, sensorData =
   example1_sensorData, engagementData =
   example1_engagementData, platformInfo =
   example1_platformInfo)
> Creating scenario ...
```

Create a scenario using create_scenario() and as much of the applicable data as you have. create_scenatio() may also give you warnings if there are problematic areas in your data, such as data dropouts. Look at these and consider then when analyzing your results.

3. Look for data dropouts

```
plot_truth_gaps(myScenario)
```

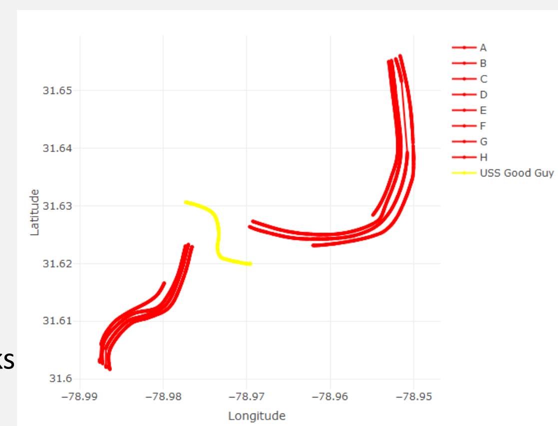
Targets A, C, and G have large gaps in data (points above the red line in the figure. Some of the functions in scenarioMaker, nautilus, and sandTable require interpolation, and these functions may not warn you if you ware interpolating through an area with limited data. In cases like these, analysts must make sure that the results through these gaps in data are still meaningful.



4a. Inspect the scenario

```
plot_truth_data_plotly(myScenario)
```

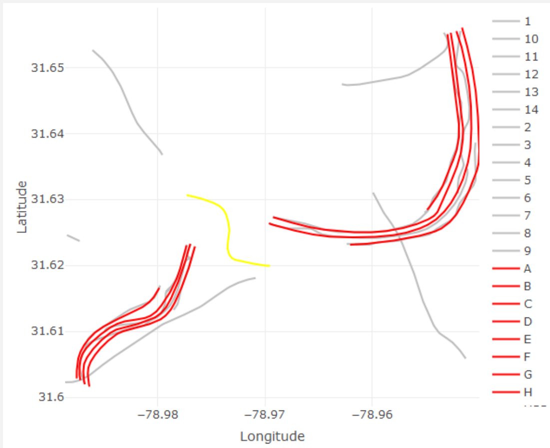
Use the plotly functions to create interactive graphs that you can inspect for completeness. You should find the areas with truth dropouts and see if they are critical regions. You should also check that the times imported correctly and that the data looks generally correct.



4b. Inspect the scenario (cont'd)

```
plot_sensor_and_truth_data_plotly(myScenario)
```

This is similar to the previous plot, but includes the sensor data. We can already see a few tracks which are probably false tracks, and we can see that some of the tracks move around between targets. We expect the target assignment in the next step to give us sensible results: tracks 1 through 9 should likely be assigned to targets while tracks 10 through 14 are false tracks.



5. Assign tracks to targets

```
myScenario <- target_assignment(myScenario, method =
   'wholeTrack', cutoff = 150) %>%
                  target_assignment_secondpass(method =
   'windowSquare', cutoff = 500, windowSize = 30)
```

Here we've chosen to use a two-pass method of target assignment. For the first pass we've chosen to use the whole track method with a smaller cutoff. This will clear out any entire tracks that are deemed false tracks. Note that if a cutoff of 100 is chosen this will cut out track 5 which looks very much like it's a valid (albeit inaccurate) track. A cutoff distance of 100 meters is probably too small for this example.

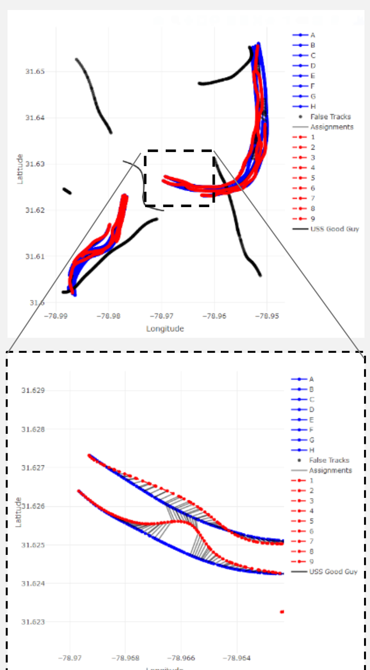
For the second pass, we use a window method with a 30 second window and a larger cutoff of 500 meters. Using a large cutoff here means that only very inaccurate points will be declared false tracks. This will probably give a better estimate of the true system accuracy than a smaller cutoff which may unfairly show the system as more accurate than it is.

6. Inspect the target assignments

```
plot_target_assignments_plotly(myScenario)
```

This figure shows which targets each track point was assigned to. In the first pass of target assignment, entire tracks were flagged as false tracks if their average distance from any target was more than 150 meters. In the second pass, the remaining points were assigned to the nearest target while averaging the distance of the sensor points within the window (time ± windowSize/2).

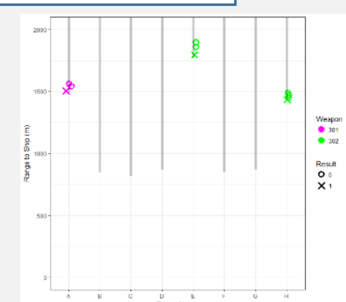
In the zoomed-in region, we see where a track was believed to switch between two targets. This looks like a reasonable explanation of the data and would not be possible if the analyst used only the whole track method.

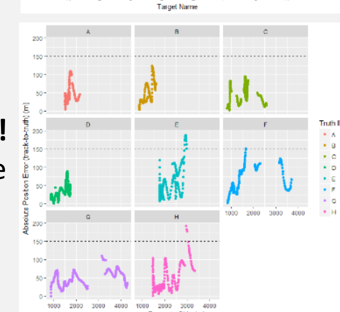


7. Analyze system performance

```
plot_money_chart(myScenario)
plot_error(myScenario, 150, doFacet = TRUE)
```

You noticed (from plot_money_chart()) that all of the kills came when targets closed within 2000 meters. You should consider looking at plot_error() to see if the error increases as a function of range to sensor platform. You should also take a look at plot_track_status() and plot_overall_coverage() (not pictured here) to see if targets are tracked better as a function of range to sensor platform.





Remember: **battleVerse automates, you analyze!** See the previous pages in the guide as well as the documentation for additional analysis functions.