ARL-TN-1306 ● APR 2026

Spatiotemporal Analysis and Inference Toolkit: Standardizing the Organization and Analysis of Multi-Agent, Human–Machine-Derived Mixed Reality Data

by Russell Cohen Hoffing, Michael Nonte, and Jonathan Touryan

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1306 ● APR 2026

Spatiotemporal Analysis and Inference Toolkit: Standardizing the Organization and Analysis of Multi-Agent, Human–Machine-Derived Mixed Reality Data

Russell Cohen Hoffing and Jonathan Touryan
DEVCOM Army Research Laboratory

Michael Nonte
DCS Corporation

DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
April 2026	Technical Note	08/01/2024	03/01/26

4. TITLE AND SUBTITLE
Spatiotemporal Analysis and Inference Toolkit: Standardizing the Organization and Analysis of Multi-Agent, Human–Machine-Derived Mixed Reality Data

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Russell Cohen Hoffing, Michael Nonte, and Jonathan Touryan

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-FC Aberdeen Proving Ground, MD 21005	ARL-TN-1306

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCIDs: Russell Cohen Hoffing, 0000-0002-3478-8196; Jonathan Touryan, 0000-0001-7343-7869

14. ABSTRACT
The Spatiotemporal Analysis and Inference Toolkit (STAIT) aims to establish a standard for 3D spatiotemporal data derived from mixed reality devices, sensors, and agents to facilitate speed and generalization of algorithm development and behavioral prediction for understanding and enhancing mixed human–agent team behavior. We provide a brief overview of STAIT background and functionality.

15. SUBJECT TERMS
Humans in Complex Systems; mixed reality; head-mounted display; human–machine integration; human–machine teaming; virtual reality; augmented reality; Information, Autonomy, and Teaming Sciences

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	54

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Russell Cohen Hoffing	(520) 691-5011

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures iv

1. Introduction 1

2. Overview of the Toolkit's Software Implementation and Functionality 2

3. Example Data Analysis and Visualizations 3

4. Conclusion 5

5. References 6

Appendix. GitLab Wiki Documentation 7

List of Symbols, Abbreviations, and Acronyms 47

Distribution List 48

List of Figures

Figure 1. Visualization of data from the repository's VR example dataset derived from Example No. 4 from the Example_Actors.m tutorial script. Black rectangles represent buildings where targets could appear. Red circles represent filtered targets that were within 10 m of the head position at any time. The blue circle represents current head position. The blue line represents the entire path of the head location. 4

Figure 2. Visualization of data from the repository's AR example dataset. (Left) Head position over the course of the dataset. Stars indicate the starting point. Quiver arrows indicate gaze direction. (Upper right) Sample by sample distance between users. (Bottom right) Gaze angle between users over time. 5

Figure A-1. Meta Quest Pro Eye data with bounding boxes extracted from VR environment. Red circles represent the possible location of targets. Black boxes represent buildings and floors. The blue circle represents head position. The yellow line represents the gaze ray. The magenta square represents the controller position. The magenta line represents the controller ray. 9

Figure A-2. Microsoft Holo Lens 2 data. Gray circles represent point cloud extracted from HL2 SLAM generated floor. Image is playback of forward facing camera data. The blue circle represents head position. The black line represents the entire path during data recording with the blue circle representing current location on that path. The magenta, green and blue arrows represent the left, forward and upward orientation vector. 10

1. Introduction

Mixed reality (MR) technologies represent a class of immersive platforms that blend elements of augmented and virtual realities (AR and VR), enabling users to perceive and interact simultaneously with virtual and physical environments. AR overlays digital information onto the physical world, maintaining high user awareness of real surroundings, typically via smartphones, tablets, or smart glasses. VR, in contrast, immerses users fully within computer-generated 3D environments, substantially reducing awareness of the physical world. Devices such as the Apple Vision Pro integrate pass-through cameras and advanced sensors to create interactive experiences where virtual objects coexist and interact with real-world elements. This spectrum of immersive technologies is characterized by varying degrees of virtuality and user engagement with physical reality, which supports rich, context-aware interactions.

To accomplish this immersive feat, modern MR devices incorporate a suite of sensor modalities—including inertial measurement units (IMUs); eye trackers; depth sensors; red, green, and blue wavelength cameras; hand tracking; and spatial audio. This sensor suite is a boon for conducting naturalistic cognitive science research by enabling high-resolution, multimodal capture of user behavior and—most importantly—simultaneously capturing environmental context. Moreover, because MR systems are built to create a unified spatial coordinate system between the world and user, scaling to multi-agent studies is feasible. This integration facilitates ecologically valid cognitive science research by allowing precise measurement of embodied cognition, visual attention, spatial navigation, decision-making, and human–machine teaming in naturalistic, ambulatory settings.[1,2] Unlike traditional laboratory-bound equipment, MR headsets provide portable, integrated sensor platforms that capture continuous spatiotemporal data streams essential for inferring cognitive states such as attention allocation. These types of experimental context and datasets are critical for developing an understanding of human behavior and technologies that generalizes to real-world settings.

Existing toolkits and frameworks in cognitive science, such as the Robot Operating System (ROS)[3] and the Brain Imaging Data Structure (BIDS)[4], have provided foundational support for data collection, organization, and analysis. ROS is widely used in robotics for sensor data collection, message passing, and coordinate frame management, offering robust middleware for real-time robotic control and simulation. However, ROS is primarily optimized for machine agents and does not natively accommodate the rich, multimodal data streams from human-worn MR devices, nor does it support the post-hoc analytical flexibility required in cognitive science research. BIDS, on the other hand, has achieved widespread adoption in

neuroimaging for organizing and describing brain imaging data, promoting reproducibility and interoperability through standardized directory structures, file-naming conventions, and metadata requirements. Despite its success, BIDS and its extensions (e.g., Motion-BIDS) are designed mainly for laboratory-based motion capture and neuroimaging data, lacking the capacity to handle the complexity of multi-agent, multi-device, MR data collected in unstructured, ambulatory environments. These limitations highlight a critical gap in existing toolkits, which focus on robotic simulation or simple, computer-based tasks and do not provide a unified world model or standardized framework for integrating heterogeneous spatiotemporal data from multiple human and robotic agents in naturalistic settings.

The Spatiotemporal Analysis and Inference Toolkit (STAIT) was developed to address these challenges by providing a standardized, extensible framework that aligns multi-agent, human and robot spatiotemporal data into a unified world model. The primary goal of STAIT is to establish a standardized data format and analytical framework that supports post-hoc analysis of spatiotemporal data collected from MR devices. STAIT provides tools for integrating diverse sensor modalities from across MR contexts such as head and eye tracking, environmental mapping like 3D point clouds, and physiological signals, thereby reducing the technical burden of data wrangling that often consumes a disproportionate share of scientists' workflows. By detailing coordinate system conventions, metadata requirements, and data schema standards inspired by BIDS and the International Society of Biomechanics' recommendations,[5] STAIT aims to enable reproducible, scalable, and generalizable analysis of complex, ambulatory mixed reality data. STAIT emphasizes flexible workflows that allow researchers to query, realign, and resegment recorded data for event and epoch analysis and tools for custom analysis. Its design facilitates the investigation of cognitive states such as attention, intention, and situational awareness in multi-agent scenarios including human–human and human–robot interactions.

2. Overview of the Toolkit's Software Implementation and Functionality

The STAIT software architecture uses an object-oriented approach and is implemented in Python and MATLAB. Data handling and storage within STAIT employs a hybrid approach to balance ease of inspection with computational performance, combining human-readable formats (e.g., JavaScript Object Notation, Comma-Separated Values) for metadata and configuration files with machine-efficient formats (e.g., Parquet) for large spatiotemporal datasets. STAIT requires input data that are spatially and temporally synchronized, or, in most cases, can be rotated or converted to have a common spatial and temporal origin across multiple

sensor modalities. Core data inputs include time-series data representing trackers (i.e., head-mounted displays) that produce 9-degrees-of-freedom position and orientation data, 3-degrees-of-freedom eye-tracking data capturing gaze origin and forward-direction orientation, and world model data representing the 3D spatial layout and object positions within the environment. Additionally, STAIT supports the incorporation of nonstandardized, custom experimental time-series data such as physiological signals (e.g., electrodermal activity and heart rate) and event markers.

Preprocessing within STAIT involves harmonizing heterogeneous data streams to a common coordinate system and temporal reference frame while accounting for differing sampling rates and missing or anomalous data points. The toolkit contains interpolation algorithms to help with data harmonization. STAIT also contains postprocessing tools, including filtering and smoothing algorithms, alongside methods for conducting event- and epoch-related analysis. For example, STAIT includes blink and saccade detection scripts in the +detect namespace (see Appendix). STAIT's most helpful postanalysis feature is the support of event-based analysis, which couple data from an individual and the environment, exemplified in Section 3. Additionally, STAIT is designed to be extensible and modular, allowing researchers to use custom functions for analysis. STAIT includes a comprehensive Wiki with example scripts and tutorials to facilitate sharing and adoption.

3. Example Data Analysis and Visualizations

To illustrate STAIT's capabilities, we include two example datasets that can be readily processed by the STAIT toolkit. Included is an example dataset collected from a VR context where a user engaged in a target detection task within a world-scale immersive 3D environment. The user shot at targets as they appeared throughout the virtual environment. The dataset includes synchronized data from a Meta Quest Pro that contains eye tracking sensors, head and hand movement sensors (i.e., embedded in the head-mounted display and via handheld controllers), custom experimental data, IMU data, and world data. Figure 1 visualizes a segment of data derived from Example No. 4 from the Example_Actors.m tutorial script.

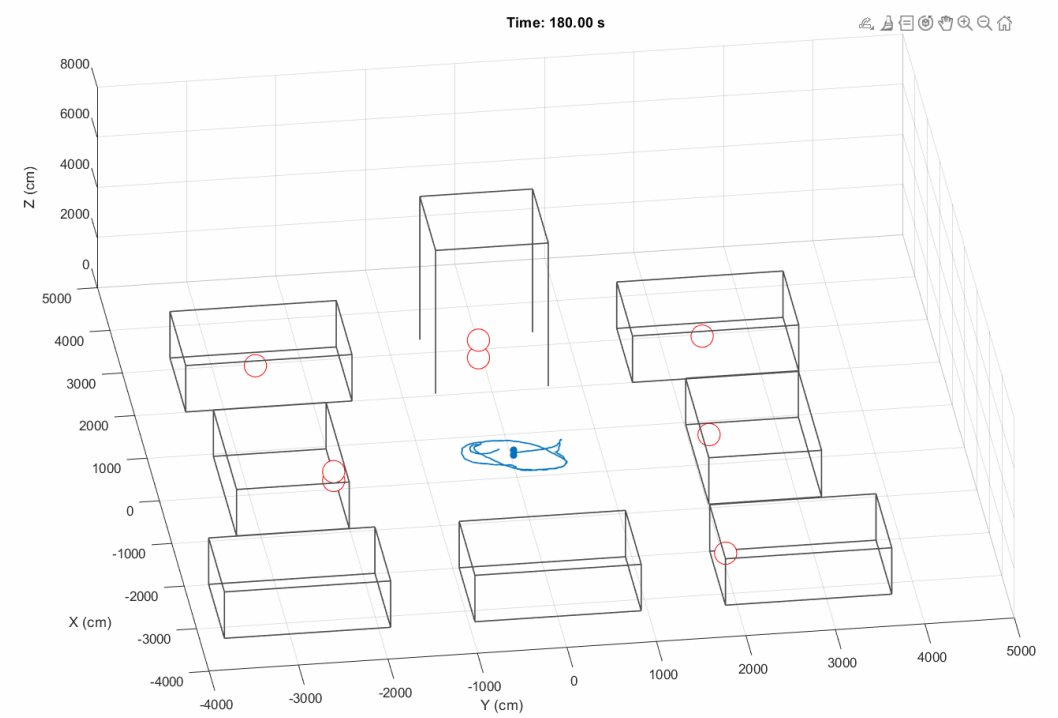


Figure 1. Visualization of data from the repository's VR example dataset derived from Example No. 4 from the Example_Actors.m tutorial script. Black rectangles represent buildings where targets could appear. Red circles represent filtered targets that were within 10 m of the head position at any time. The blue circle represents current head position. The blue line represents the entire path of the head location.

In a separate example AR dyadic dataset, two users wearing Microsoft HoloLens 2 headsets engaged in a collaborative target detection task. Figure 2 shows STAIT's functionality for analyzing and revealing patterns of joint attention and coordination dynamics.

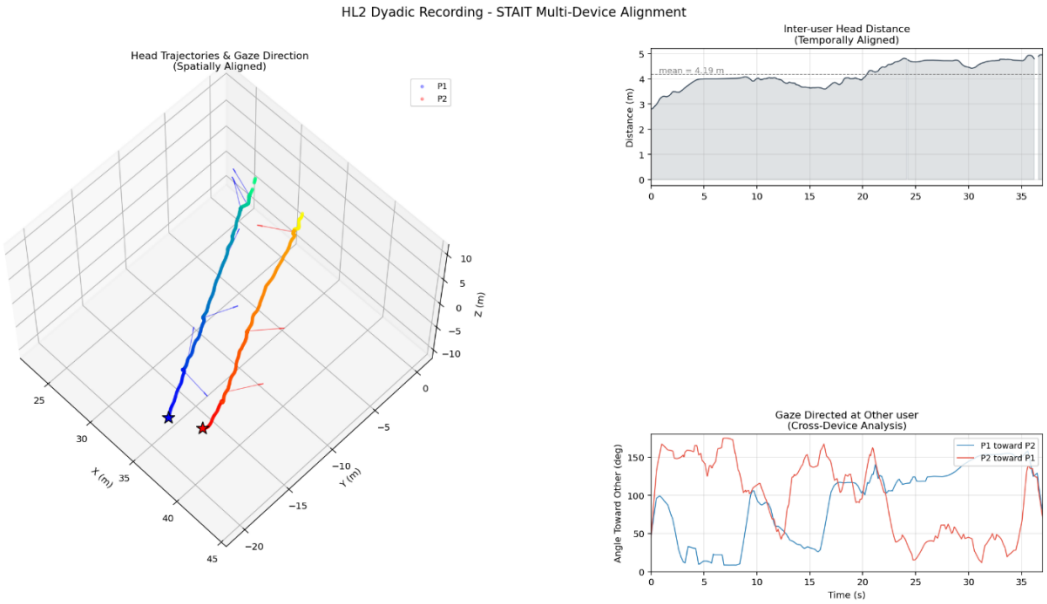


Figure 2. Visualization of data from the repository's AR example dataset. (Left) Head position over the course of the dataset. Stars indicate the starting point. Quiver arrows indicate gaze direction. (Upper right) Sample by sample distance between users. (Bottom right) Gaze angle between users over time.

4. Conclusion

STAIT addresses critical gaps in the analysis of spatiotemporal data for cognitive science research involving MR devices by providing a standardized, extensible framework that integrates heterogeneous multimodal data into a unified world model. This approach addresses limitations of existing toolkits that are either optimized for robotic simulation or constrained to simple, laboratory-based tasks. Additionally, it enables rigorous posthoc analysis of complex, ambulatory, multi-agent data collected in naturalistic environments. By harmonizing coordinate systems, creating data and metadata standards, and providing tools for pre- and postprocessing, STAIT facilitates reproducibility, scalability, and cross-study comparability, which are essential for enabling behavioral insights in naturalistic contexts.

5. References

1. Maselli A et al. Beyond simple laboratory studies: developing sophisticated models to study rich behavior. Physics of Life Reviews. 2023;46:220–244.
2. Goettker A, Powell N, Hayhoe M. Approaches to understanding natural behavior. Journal of Vision. 2025;25(6):12.
3. Macenski S, Foote T, Gerkey B, Lalancette C, Woodall W. Robot operating system 2: design, architecture, and uses in the wild. Science Robotics. 2022;7(66):eabm6074
4. Gorgolewski KJ et al. BIDS apps: improving ease of use, accessibility, and reproducibility of neuroimaging data analysis methods. PLOS Computational Biology. 2017;13(3):e1005209.
5. Cereatti A et al. ISB recommendations on the definition, estimation, and reporting of joint kinematics in human motion analysis applications using wearable inertial measurement technology. Journal of Biomechanics. 2024;173:112225.

Appendix. GitLab Wiki Documentation

This appendix appears in its original form, without editorial change.

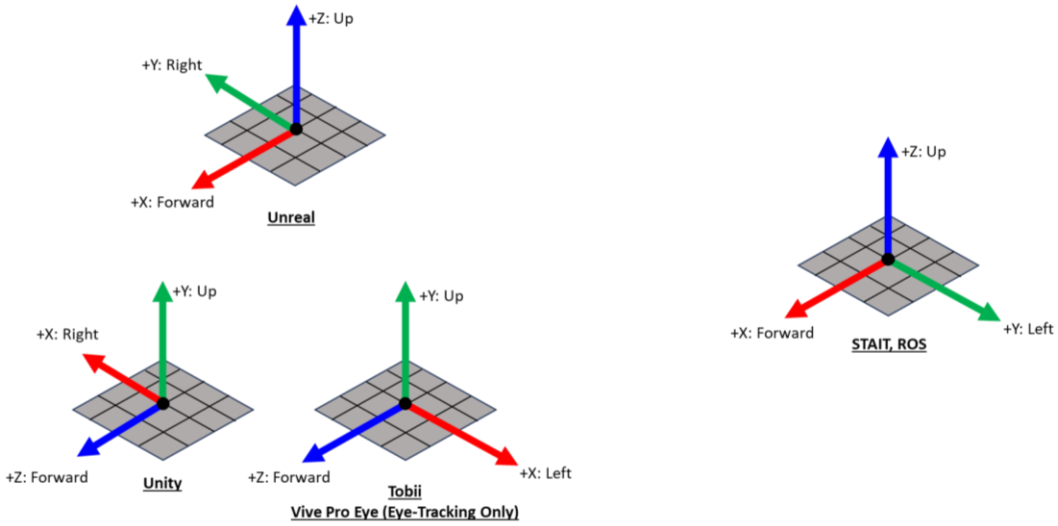
A.1 Home.md

Welcome to the **Spatio-Temporal Agent Inference Toolkit (STAIT)** Wiki!

Here you'll find information on the STAIT standard and toolkit. Use the sidebar to navigate to different sections starting with the README.

What problem does STAIT try to tackle?

In short:



These coordinate systems represent the axis of 3D location data (e.g. position and orientation) from various virtual environments and VR/AR/XR devices and sensors (e.g. eye tracker, IMU). As you can see, a positive Y axis value does not mean the same thing for all sources of data, because there is no standardization of coordinate systems across sensors and devices. STAIT aims to create a standardized data format to capture movement and perspective in space and over time (i.e. spatiotemporal data). We are tackling this problem from a multi-agent, mixed-reality perspective where multiple individuals and robots, with different sensors and effectors, exist in the same space and time. We seek to standardize this data, not only for ease of data analysis, but ultimately to enable generalizable algorithm development for agents that sense and act in the world. While there are related frameworks, tools, and resources in the field of autonomy - none exist that can represent the unique aspects of **both humans and autonomous systems**.

We take inspiration from the Brain Imaging Data Standard (BIDS), specifically Motion-BIDS, as well as the International Society of Biomechanics' recommendations for standardization in the reporting of kinematic data. We also leverage the numerous tools and approaches used in the computer vision, spatial computing, and autonomy communities.

Who are the people we made this for? We are making STAIT for researchers like us who conduct experiments and analyze 3D spatiotemporal data in ambulatory contexts from a range of devices. On the human-sensing side, we use devices such as head-mounted displays (e.g., Quest, HTC Vive, HoloLens, Varjo), mobile eye-trackers (e.g., Tobii Glasses 3, Pupil Labs), and IMUs (e.g., Noraxon). On the autonomy side, we use a range of systems from small unmanned areal systems to legged robotics. We are cognitive scientists and care about measuring where what and when people are looking/moving/responding, and using that data to infer states of attention, intention, and awareness.

We want to make it easy to create visualizations like this:

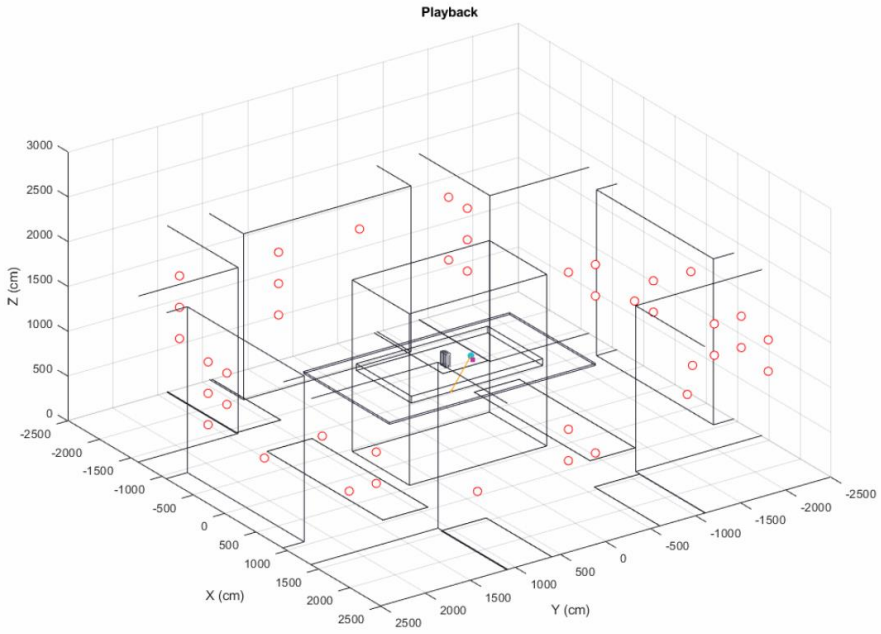


Figure A-1. Meta Quest Pro Eye data with bounding boxes extracted from VR environment. Red circles represent the possible location of targets. Black boxes represent buildings and floors. The blue circle represents head position. The yellow line represents the gaze ray. The magenta square represents the controller position. The magenta line represents the controller ray.

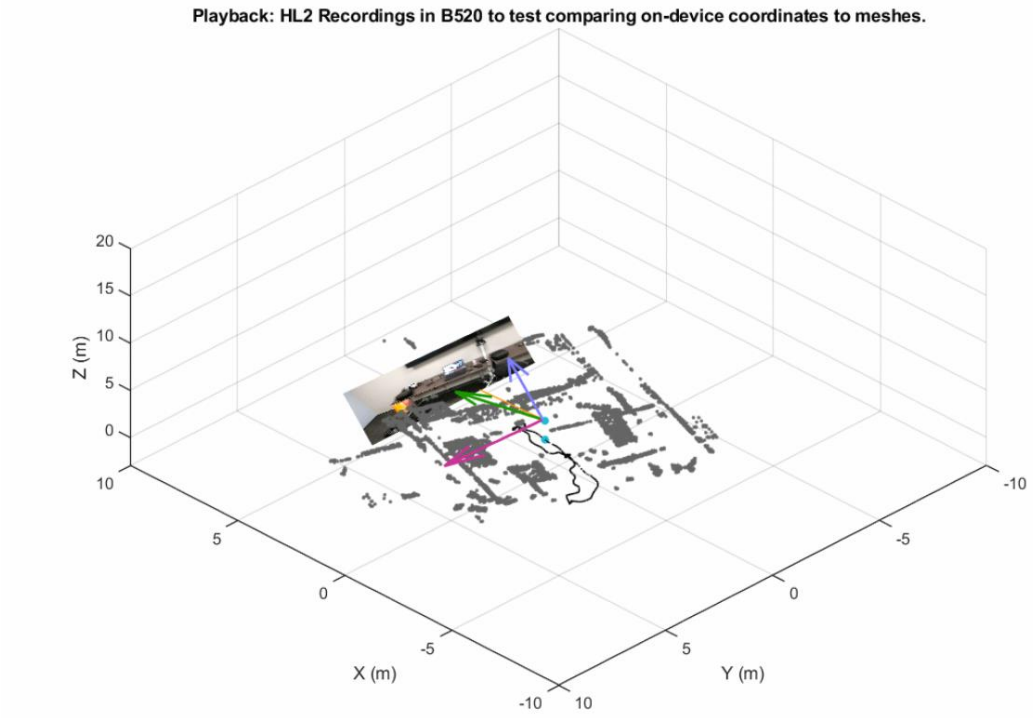


Figure A-2. Microsoft Holo Lens 2 data. Gray circles represent point cloud extracted from HL2 SLAM generated floor. Image is playback of forward facing camera data. The blue circle represents head position. The black line represents the entire path during data recording with the blue circle representing current location on that path. The magenta, green and blue arrows represent the left, forward and upward orientation vector.

A.2 Home\README.md

Overview of STAIT

The toolkit aims to abstract and standardize the types of data that would be captured from mixed reality contexts including perspective (i.e. 1st person viewpoint from a mounted camera), position (i.e. from a VR controller), orientation (i.e. from IMUs), or world data (i.e. 3D mesh or VR actors list).

We currently target the following COTS devices and platforms:

- Meta Quest Pro
- HoloLens 2
- Noroxon IMU

- Unreal engine
- ROS
- Microsoft PSI

We use an object-oriented approach that defines a class hierarchy with standardized properties and methods to enable easy and generalizable processing of data. We represent objects and agents from the world using the Actor class and time-series data from those actors using the TimeSeries class.

Class Hierarchy

The TimeSeries object is the most generic container that allows for loading, manipulating, and simple preprocessing (e.g. artifact detection) of any timeseries-type data. The second tier of objects inherits class properties (attributes and methods) specific to the defined data type. Our current classes include TimeSeries, GazeOriginSeries, TrackerSeries, TimeSeriesBuilder and TimeSeriesCollection. We are currently developing additional timeseries classes such as IMU- stay tuned.

The Actor Class, ActorBuilder, and ActorCollection allows us to capture information about agents and objects in the world. While you can use the TimeSeries class independently from the Actor class, this class is useful for conducting analysis in relation to objects and agents in the world such as visualizations or creating object-related events or epochs (e.g. proximity to an object).

We leverage the Unreal Engine plugin, UETack, for VR data. UETack supports any VR headset that can be powered by an Unreal application (e.g. Meta Quest Pro, Varjo XR series). The UETack pipeline unreal plugin provides a common schema for capturing VR data, including timeseries elements (e.g. camera position) and world elements (e.g. actors). We have also developed wrangling/importing tools from other common formats: LSL/XDF, PSI and ROS.

A.2.1 Raw

While we will provide wrangling code examples for specific systems and recording platforms (e.g. UETack + Quest Pro), this code will necessarily be task and configuration specific. Our wrangling code should be used as an example for how to import and organize the data into the STAIT format. See Requirements and Schemas documentation for ensuring compatibility with STAIT.

Wrangling may require transforming, rotating and converting into the STAIT standard and be assigned required properties including metadata, IsMissing table, IsAnamoulous table, and the DataAttributes table.

A.2.2 Level 0 - "All Spatial Elements, All in the Same Space!"

In Level 0 data has been aligned spatially, has a common temporal origin. Note in our example data, we export all relevant data from the raw data. Here there is a mix of timeseries, discrete data (i.e., actors) and custom data that contains idiosyncratic information about the experiment. This data is read into MatLab/Python using non-STAIT tools like parquetread because it doesn't have the IsMissing data table assigned yet.

A.2.3 Level 1 - "All Temporal Streams, All in the Same Time!".

In Level 1 only timeseries data that conforms to standards is temporally aligned and assigned properties like attributes. Attributes define the appropriate methods that are applied to this type of data when using STAIT built-in functionality. For example, if data is circular versus linear, than data must be unfolded to allow for proper interpolation. This step also includes upsampling/downsampling and interpolation and assigning the IsMissing table for interpolated data. Level01 ensures all data have the same row timestamps, making it easy to align, analyze, and visualize behavior over time. Having data row aligned is meant to make Level 2 data analysis easy.

A.2.4 Level 2

Now that the data is in the same space and time, Level 2 is where analysis of behavior happens. The toolkit supports a number of methods including event/epoch based analysis and time series based analysis. We use a python-inspired namespace framework to represent the different types of processing functions.

A.3 Home\Developers.md

This toolkit was made with two end users in mind, the 'developer' and the 'researcher'. The developer is the individual who is programming the experiment,

has knowledge of the data collection pipeline, sensors and will be ensuring compatibility with STAIT requirements. The researcher is the individual who designed the research question, experiment and will be conducting analysis by leveraging the STAIT toolkit. The researcher or the developer may be responsible for transforming or converting coordinates to comply with STAIT standards.

For STAIT compatibility requirements and standards see Requirements and Schemas.

The toolkit is designed to be leveraged by individuals who don't have extensive experience with 3D coordinate systems in extended reality, but is flexible enough to allow for custom development. For example, given needs of a lab, a developer may implement often used functions for a particular type of sensor like an inertial measurement unit (IMU) or GPS. See Processors for how to develop processors.

A.4 Home\Developers\Schemas.md

Coordinates are expressed in world coordinates, where during the data collection process a common spatial origin is defined, and during wrangling all sensor coordinates are transformed relative to the spatial origin. X+ coordinate values corresponds to the forward direction, Y+ coordinate values corresponds to the left direction, and Z+ coordinate values corresponds to the up direction.

During wrangling the following properties should be defined for every level01 data stream. Here is an example of level 01 data TrackerSeries_Example_Expected_Data.csv (converted to a .csv file) showing expected data and format. As a note `IsAnomalous` is not required to be defined during wrangling and upon import will be automatically generated with false values matching the size of the TimeSeries data (minus the time column).

- `Data`
- `Metadata` - The TimeScaler field is currently required.
- `IsMissing`
- `DataAttributes` - Required are name, unit manifold and domain fields. Manifolds represent the structure or type of data representation. Currently only linear, circular and spherical manifolds are supported. String data and Boolean data is not currently supported. Domain represent the range of data values and need to contain two numeric values with the second number greater in value than the first.

Meta Data example:

vr_gaze_main.json

For Actors the following properties should be defined: Actors have to have an Attribute called ‘Name’. Possessions and Owner is a struct. State will be a table, Attributes is a struct.

For expanded file format, here is an example of the ActorsExampledirectory_tree.txt. Future changes will include an HDF format.

Gaze Origin Series data description:

Gaze Origin Series

Variable	Data Type	Units	Description
Time	Int64	ms	Time represented as an integer in milliseconds
Location_x	Float64	cm	The location of the gaze origin on the X-axis
Location_y	Float64	cm	The location of the gaze origin on the Y-axis
Location_z	Float64	cm	The location of the gaze origin on the Z-axis
Direction_x	Float64	unit vector	X-axis direction of the gaze origin
Direction_y	Float64	unit vector	Y-axis direction of the gaze origin
Direction_z	Float64	unit vector	Z-axis direction of the gaze origin

Tracker Series data description:

Tracker Series

Variable	Data Type	Units	Description
Time	Int64	ms	Time represented as an integer in milliseconds
Location_x	Float64	cm	The location of the object on the X-axis
Location_y	Float64	cm	The location of the object on the Y-axis
Location_z	Float64	cm	The location of the object on the Z-axis
Forward_x	Float64	unit vector	The orientation of the Forward vector along the X-axis
Forward_y	Float64	unit vector	The object orientation of the Forward vector along the Y-axis
Forward_z	Float64	unit vector	The object orientation of the Forward vector along the Z-axis
Left_x	Float64	unit vector	The object orientation of the Left vector along the X-axis
Left_y	Float64	unit vector	The object orientation of the Left vector along the Y-axis
Left_z	Float64	unit vector	The object orientation of the Left vector along the Z-axis
Up_x	Float64	unit vector	The object orientation of the Up vector along the X-axis
Up_y	Float64	unit vector	The object orientation of the Up vector along the Y-axis
Up_z	Float64	unit vector	The object orientation of the Up vector along the Z-axis

A.5 Home\Developers\Requirements.md

Necessary Requirements to use STAIT

- Temporal data streams (e.g. timeseries, events, etc.) should use a single reference clock or information about the relative alignment between multiple clocks. Refer to LSL, UETack, PSI, Open XR for examples.
- Spatial data streams (e.g. location, position, orientation) should be represented relative to a common (global) spatial origin and heading (e.g. location when recording was started). These can be relative spatial reference frames or geospatial reference frames. Here is an example of the metadata collected for one of our data sets. Note the Origin location and heading was captured which describes the origin (0,0,0) for this data set and was captured using the following calibration protocol used to validate data across multiple devices.
- Metadata describing the configuration (devices, coordinate system, channels, subject info, etc.). Note this step can happen post-data collection during wrangling as in our wrangling example.

STAIT Dependencies Currently, Python is used for wrangling data. See requirements for wrangling code. Analysis code has been mostly developed in MatLab.

Recommended not required * Capture actor information. We use UETack to capture actor information in Unreal environments.

A-6 Home\Developers\Processors.md

Custom processor information coming soon!

A.7 Definitions

STAIT coordinate system - X+ coordinate values corresponds to the forward direction, Y+ coordinate values corresponds to the left direction, and Z+ coordinate values corresponds to the up direction.

STAIT toolbox specific definitions - throughout the wiki we use the following terms.

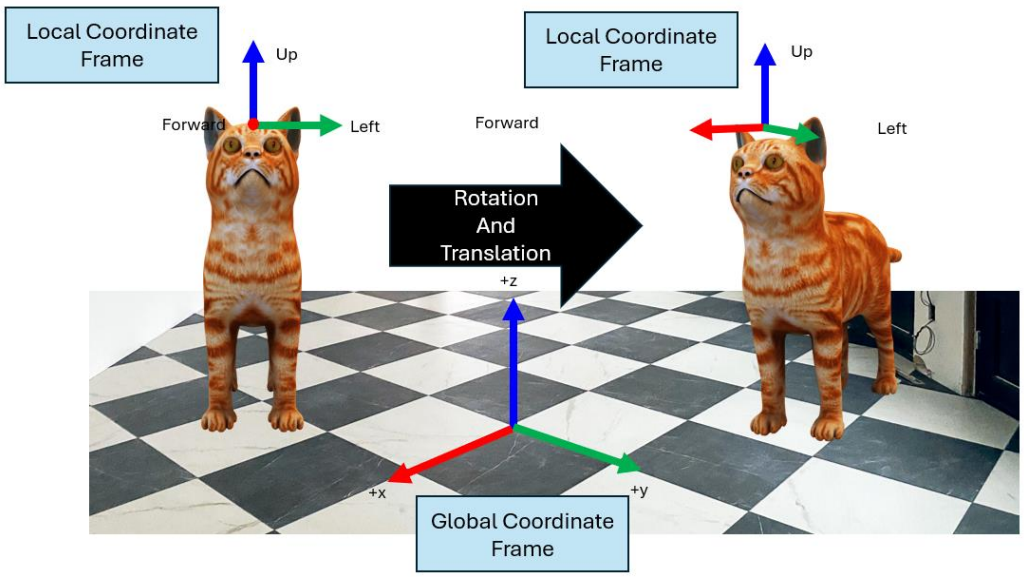
`TimeSeries object` - This refers to all objects created using the STAIT toolbox classes TimeSeries, GazeOriginSeries or TrackerSeries.

`Properties` - This refers to everything that defines a class including the Data, MetaData, Attributes, etc.

World Coordinates - Coordinates defined relative to the experimental space. All STAIT coordinates are expressed in world coordinates.

Global Coordinates - Coordinates defined relative to geomagnetic earth.

Local Coordinates - Coordinates defined relative to the sensor.



Definition Figure

A.8 FAQ

1. Why are we using classes and objects?
   A: We are aiming to develop a spatiotemporal standard that will function across a myriad number of devices and data structures. Classes allow us to strictly define data structures such that functions and operations can be applied easily once contained within a class object regardless of the type of data. Additionally, it keeps most of the functionality hidden so that users can use this code in a more plug and play manner.

2. What is your experimental workflow using STAIT
   A: Given the technical skills required for extended reality simulations, we typically have one person who is doing the programming for the experiment who has knowledge of the sensors and backend for exporting that data. This 'developer' is the person who will ensure that all STAIT compatible requirements and data are achieved and exported. The developer will also be the one who will have the knowledge to transform or convert raw data into level00 or level01 data. After conversion, once STAIT data is available, a researcher uses the toolbox to perform analysis of the data.

A.9 Git-Quick-Reference.md

Common Git Commands (for this repo)

If you just want to use STAIT, start with:

`>git clone https://gitlab.sitcore.net/VisualSearch/STAIT.git`

Then you just need to check for updates from time-to-time:

`>git pull`

If you want to develop, use the following:

`>git checkout branchname` (select or switch your branch)

`>git checkout -b mybranch` (or create a new one)

Always check the status of your branch before editing and use the most recent version:

`>get branch`

`>git fetch`

`>git status`

`>git pull`

Then edit files and/or add new ones. When you're done:

`>git add myfile.m`

`>git commit -a` (for long comments & large commits)

`>git commit -m "this is what I changed/added"` (for simple commits)

`>git push origin mybranch`

Then use the web interface for a merge request. When the merge is completed, clean up:

`>git branch -d mybranch` (to delete a merged branch)

`>git branch -D mybranch` (to force delete an unmerged branch)

Basic GIT Commands

Command	Explanation
git clone [url]	Copies the git repository from gitlab
git status	Shows which files are changed, added, or removed
git log	Shows a log of commits for current branch
git fetch	Reaches out to GitLab for updates, but doesn't apply them
git add [list of file names]	Starts tracking file(s). Use –A or "." to add all modified files
git commit –m "[message]"	Commits added files to the current branch. Omit –m flag to add longer messages.
git pull origin [remote branch name]	Fetches and merges a branch from GitLab
git push origin [branch name]	Sends the committed branch to GitLab
git remove [branch name(s)]	Removes merged branches from local repository. –D flag will allow you to delete branches that haven't been merged

gitCommands

A.10 Home\Classes.md

Documentation of classes (and types of sensor or experimental data) currently supported by the toolkit.

TimeSeries - All *Series classes inherit TimeSeries methods TimeSeriesBuilder TimeSeriesCollection GazeOriginSeries TrackerSeries Actor ActorCollection ActorBuilder Actor Code

A.11 Home\Classes\TimeSeries.md

The `TimeSeries` class is the top-level superclass for STAIT timeseries data. All other STAIT timeseries classes inherit from `TimeSeries`. Properties and methods are accessed using dot notation, e.g., `TS.<Property or Method>`, where `TS` is the variable name of the `TimeSeries` object.

A.11.1 Table of Contents

- Properties
  - .Data
  - .Metadata
  - .IsAnomalous
  - .IsMissing
  - .DataAttributes
  - .History
  - .ConstructorHandle

- Methods
    - .check_standard()
    - .concat(ts_to_add, dim)
    - .describe()
    - .drop(col_names)
    - .keep(col_names)
    - .mask_missing()
    - .pipe(function_handle, varargin)
    - .read_stait(fpath)
    - .slice_time(start_time, stop_time)
    - .write_stait(fpath)

A.11.2 Properties

Properties define the rules and data structures for the STAIT standard and toolbox. These are accessed via dot notation but cannot be directly assigned to, as this would violate the object's integrity.

`.Data` A MATLAB `table` containing the time-aligned data. Columns within `.Data` can be accessed directly from the object without referencing `.Data`. For example, `TS.location_x` is equivalent to `TS.Data.location_x`.

`.Metadata` A `struct` containing metadata extracted from the associated `.json` file. This includes information about the data collection device, temporal and spatial origins, and other experimental details.

`.IsAnomalous` A `table` with the same dimensions as `.Data` (excluding the 'time' column) containing values of `1` or `NaN`. A value of `1` indicates that the corresponding data point is considered anomalous, while `NaN` indicates a non-anomalous point. This table is inherited when new `TimeSeries` objects are created from an existing one, ensuring anomaly information is preserved.

`.IsMissing` A `table` with the same dimensions as `.Data` (excluding the 'time' column) containing values of `1` or `NaN`. A value of `1` indicates a missing data point, while `NaN` indicates a non-missing point. This table is inherited when new `TimeSeries` objects are created, ensuring missing data information is preserved.

`.DataAttributes` A `table` containing metadata for each column in `.Data`. This includes variable names, units, data manifolds (e.g., linear, circular), and domains (e.g., the expected range of values).

`.History` Tracks the history of data transformations and methods applied to the `TimeSeries` object. Note: This functionality is not fully implemented in the current version.

`.ConstructorHandle` Stores the constructor class handle for the `TimeSeries` object (e.g., `@TimeSeries`, `@TrackerSeries`). This allows methods that create a new object to return an object of the same class.

A.11.3 Methods

`.check_standard()` Checks if the `TimeSeries` object conforms to the required standard, such as ensuring a 'time' column is present. Returns `true` if the object is valid.

`.concat(ts_to_add, dim)` Concatenates two `TimeSeries` objects. * **`ts_to_add`**: The `TimeSeries` object to concatenate with the current object. * **`dim`**: The dimension for concatenation. Currently, only horizontal concatenation (`dim = 2`) is supported. Vertical concatenation (`dim = 1`) is not yet implemented. * **Behavior**: If there are duplicate column names between the two objects, the columns from `ts_to_add` are renamed by appending an underscore (_) to avoid conflicts. The method merges the `Data`, `IsMissing`, `IsAnomalous`, and `DataAttributes` tables. * **Returns**: A new, concatenated `TimeSeries` object.

`.describe()` Generates and displays a summary `table` describing the data columns in the `TimeSeries` object. The summary includes statistics such as sample count, percent missing, percent anomalous, mean, standard deviation, minimum, quartiles, and maximum for each column.

`.drop(col_names)` Removes specified columns from the `TimeSeries` object. * **`col_names`**: A cell array of strings containing the names of the columns to be removed. * **Note**: The `time` column cannot be dropped. * **Returns**: A new `TimeSeries` object without the specified columns.

`.keep(col_names)` Keeps only the specified columns in the `TimeSeries` object, discarding all others. * **`col_names`**: A cell array of strings containing the names of the columns to be kept. * **Note**: The `time` column is always kept automatically, even if not specified in `col_names`. * **Returns**: A new `TimeSeries` object with only the specified columns.

`.mask_missing()` Replaces data points flagged in the `.IsMissing` table with `NaN` in the `.Data` table. This is useful for ensuring that missing data points are not included in calculations. If the `.IsMissing` table is not populated, a warning is issued. * **Returns**: A new `TimeSeries` object with missing data masked.

`.pipe(function_handle, varargin)` Applies a user-defined function to the `TimeSeries` object, allowing for custom, chainable operations. See custom function tutorial for reference. * **`function_handle`**: A handle to a function that accepts a `TimeSeries` object as its first argument. * **`varargin`**: Optional additional

arguments to be passed to the function. * **Note**: The provided function must return a `TimeSeries` object.

`.read_stait(fpath)` A static method that reads a STAIT-formatted `.pqt` file and its associated `.json` metadata file from a specified path. * **`fpath`**: A string representing the file path and name, without the extension. * **Returns**: A `TimeSeries` object populated with the data from the files.

`.slice_time(start_time, stop_time)` Extracts a temporal slice of the `TimeSeries` object based on a specific start and stop time. * **`start_time`**: The start time for the slice. * **`stop_time`**: The stop time for the slice. * **Returns**: A new `TimeSeries` object containing only the data within the specified time interval.

`.write_stait(fpath)` Writes the `TimeSeries` object to a STAIT-formatted `.pqt` file and its metadata to a `.json` file. * **`fpath`**: A string representing the file path and name for the output files, without an extension. * **Behavior**: * The `.pqt` file contains the columns from `.Data`. If `.IsMissing` and `.IsAnomalous` tables exist, their columns are appended to the output table. Columns originating from `.IsMissing` are prefixed with `is_missing_`. * The `.json` file contains the contents of `.Metadata`, with the `.DataAttributes`

A.12 Home\Classes\TimeSeriesBuilder.md

TimeSeriesBuilder Code

The `TimeSeriesBuilder` is a utility class that implements the **Builder design pattern**. Its purpose is to separate the complex process of constructing a `TimeSeries` object from the `TimeSeries` class definition itself. This provides a flexible, step-by-step, and readable way to create new `TimeSeries` objects, whether from scratch, from files, or by modifying existing objects. It is most useful when creating custom analysis functions as custom functions must input and output `TimeSeries` objects.

A.12.1 Table of Contents

- Properties
- Methods
  - Builder Methods
    - .copy(ts)
    - .read_stait(fpath)
  - Instance Methods
    - .add_Data(Data)
    - .add_Metadata(Metadata)

- .add_IsAnomalous(IsAnomalous)
- .add_IsMissing(IsMissing)
- .add_DataAttributes(DataAttributes)
- .add_History(History)
- .add_ConstructorHandle(ConstructorHandle)
- .build(constructor_handle)

A.12.2 Properties

The `TimeSeriesBuilder` object holds the properties (`Data`, `Metadata`, `IsAnomalous`, etc.) that will be used to construct a new `TimeSeries` object. These properties are identical in structure to those found in the TimeSeries class and are populated using the `add_*` methods described below.

A.12.3 Methods

The methods are divided into two categories: builder methods, which create a new builder instance, and instance methods, which configure the builder before creating the final object.

Builder Methods These methods are the starting points for creating and pre-populating a `TimeSeriesBuilder` instance. They are called directly from the class name (e.g., `TimeSeriesBuilder.copy(ts)`). To initialize a `TimeSeriesBuilder` object use the syntax `TSB=TimeSeriesBuilder()`.

`.copy(ts)` Creates a new `TimeSeriesBuilder` instance by copying all the properties from an existing `TimeSeries` object (`ts`). This is the primary method for creating a builder when you intend to modify an existing object. * **`ts`**: An existing `TimeSeries` object (or any of its subclasses). * **Returns**: A new `TimeSeriesBuilder` instance pre-populated with the properties from `ts`.

`.read_stait(fpath)` Constructs a `TimeSeriesBuilder` instance by reading and parsing STAIT-formatted files. It reads the `.pqt` file for data and the `.json` file for metadata, automatically separating the `Data`, `IsMissing`, and `DataAttributes` properties. * **`fpath`**: A string representing the file path and name, without the extension. * **Returns**: A new `TimeSeriesBuilder` instance populated with data from the specified files.

Instance Methods These methods are called on an instance of the `TimeSeriesBuilder` to add or modify its properties. They are designed to be **chained** together for fluid and readable object construction.

`.add_Data(Data)` Adds or replaces the `Data` table in the builder. It validates that the input is a `table`, contains a `time` column, and all columns are numeric. * **Returns**: The `TimeSeriesBuilder` object, allowing for method chaining.

`.add_Metadata(Metadata)` Adds or replaces the `Metadata` struct in the builder. It validates that the struct contains a numeric `Timescaler` field. * **Returns**: The `TimeSeriesBuilder` object.

`.add_IsAnomalous(IsAnomalous)` Adds or replaces the `IsAnomalous` table. It validates that the input is a `table` containing only `0`s and `1`s and casts it to a logical type. * **Returns**: The `TimeSeriesBuilder` object.

`.add_IsMissing(IsMissing)` Adds or replaces the `IsMissing` table. It validates that the input is a `table` containing only `0`s and `1`s and casts it to a logical type. * **Returns**: The `TimeSeriesBuilder` object.

`.add_DataAttributes(DataAttributes)` Adds or replaces the `DataAttributes` table. It performs extensive validation to ensure the table contains the required rows (`Units`, `Manifold`, `Domain`) and columns (`time`), and that the values for `Manifold` and `Domain` are valid. * **Returns**: The `TimeSeriesBuilder` object.

`.add_History(History)` Adds or replaces the `History` property. (Note: Validation for this property is a placeholder in the current version). * **Returns**: The `TimeSeriesBuilder` object.

`.add_ConstructorHandle(ConstructorHandle)` Adds or replaces the `ConstructorHandle`. It validates that the handle (e.g., `@TimeSeries`, `@TrackerSeries`) is a recognized constructor. * **Returns**: The `TimeSeriesBuilder` object.

`.build(constructor_handle)` This is the final method in the builder chain. It assembles all the added properties into a new `TimeSeries` object.

- **`constructor_handle`** (optional): A function handle for the class to be constructed (e.g., `@TimeSeries`, `@TrackerData`, or `@GazeData`). If provided, it overrides any handle already in the builder.
- **Behavior**:

    1. **Validation**: Before building, it checks that all required properties (`Data`, `Metadata`, `DataAttributes`) have been added. It also verifies that the column names are consistent across the `Data`, `DataAttributes`, `IsMissing`, and `IsAnomalous` tables.

2. **Auto-Population**: If optional properties like `IsAnomalous` or `IsMissing` have not been added, it automatically generates and adds default tables filled with `false` values.
3. **Construction**: It calls the constructor of the target class using the provided `constructor_handle` (or one stored in the builder) and passes in all the collected properties.
4. **Final Check**: It runs the `.check_standard()` method on the newly created `TimeSeries` object to ensure it complies with the data standard.

- **Returns**: A new, fully validated `TimeSeries` (or subclass) object. TimeSeriesCollection Code

A.13 Home\Classes\TimeSeriesCollection.md

The `TimeSeriesCollection` class acts as a container for managing multiple `TimeSeries` objects. It stores a group of related `TimeSeries` objects in a single collection, allowing for compact representation and, most importantly, batch processing.

The primary feature of this class is its ability to apply the same operation to every `TimeSeries` object within the collection using a single command. This is particularly useful when identical processing steps, such as resampling or filtering, need to be performed on a set of related data streams (e.g., head tracking and gaze data).

A.13.1 Table of Contents

- Properties
  - .Collection
- Methods
  - .TimeSeriesCollection()
  - .add(ts)
  - .drop(idx_drop)
  - .pipe(fn_handle, varargin)

A.13.2 Properties

The `TimeSeriesCollection` has one primary property to store its contents that are created using the methods below as this can't be directly assigned. TimeSeries objects contained within the collection are accessed via the `TSC.Collection{x}` syntax.

`.Collection` A cell array that holds the individual `TimeSeries` objects. Each element in the cell array is a complete `TimeSeries` object (or an object of a `TimeSeries` subclass like `TrackerData` or `GazeData`).

A.13.3 Methods

The methods for `TimeSeriesCollection` are focused on managing the contents of the collection and applying batch operations. Like the `TimeSeries` class, its methods are designed to be immutable, meaning they return a *new*, modified `TimeSeriesCollection` object rather than changing the original one.

`.TimeSeriesCollection()` The constructor method used to create a new, empty `TimeSeriesCollection` object. The `.Collection` property is initialized as an empty cell array.

`.add(ts)` Adds a `TimeSeries` object to the collection. The method validates that the input is a `TimeSeries` object (or a subclass) before adding it. This method can be chained to add multiple objects fluidly.

- **`ts`**: The `TimeSeries` object to be added to the collection.
- **Returns**: A new `TimeSeriesCollection` object containing the newly added element.

`.drop(idx_drop)` Removes one or more `TimeSeries` objects from the collection based on their index within the `.Collection` cell array.

- **`idx_drop`**: An integer or an array of integers specifying the index (or indices) of the objects to remove.
- **Returns**: A new `TimeSeriesCollection` object with the specified elements removed.

`.pipe(fn_handle, varargin)` Applies a function to **every** `TimeSeries` object within the collection. This method iterates through each object in `.Collection` and calls its respective `.pipe()` method, passing the function handle and any additional arguments. This is the core feature for batch processing.

- **`fn_handle`**: A handle to a function that accepts a `TimeSeries` object as its first argument and returns a `TimeSeries` object.
- **`varargin`**: Optional additional arguments to be passed to the function specified by `fn_handle`.
- **Returns**: A new `TimeSeriesCollection` object where each internal `TimeSeries` has been replaced by the result of the piped function.

A.14 Home\Classes\Actor.md

The `Actor` class represents any discrete entity in an experiment, such as objects, human participants, or non-human agents. This class aggregates `TimeSeries` data relevant to the entity, along with its permanent (`Attributes`) and transient (`State`) properties. Properties and methods are accessed using dot notation, e.g., `MyActor.<<>Property or Method>`, where `MyActor` is the variable name of the `Actor` object.

A.14.1 Table of Contents

- Properties
  - .Attributes
  - .State
  - .Possessions
  - .Owner
  - .Data
- Methods
  - .read_expanded(fpath)
  - .add_possession(new_possession)
  - .drop_possession(possession_name)
  - .add_owner(owner, name_owner)
  - .to_table()

A.14.2 Properties

Properties define the data structures that describe an `Actor` object. These are accessed via dot notation.

`.Attributes` A `struct` containing properties of the actor that do not change over time. This typically includes static information like a unique identifier (`id`), type, or physical characteristics.

`.State` A `struct` containing properties of the actor that can change over time. This can include transient states such as visibility, position, or status.

`.Possessions` A `struct` containing names of objects that are possessed by this actor, creating a hierarchical relationship.

`.Owner` A `struct` containing the `Actor` object(s) that possess this actor. This provides a reverse link to the `Possessions` property, allowing for traversal up the hierarchy.

`.Data` Contains a `TimeSeriesCollection` object that describe the actor's behavior and measurements over time. This property holds the dynamic data associated with the actor.

A.14.3 Methods

Methods are functions that operate on or are related to `Actor` objects.

`.read_expanded(fpath)` A static method that reads actor data from an "expanded" directory format and constructs an `Actor` object. * **`fpath`**: A string representing the file path to the actor's folder. This folder is expected to contain a subfolder of State tables (`.pqt`) and a `.json` file containing the Attributes. * **Behavior**: This method utilizes `ActorBuilder.read_expanded(fpath).build()` to construct the object. * **Returns**: A new `Actor` object populated with data from the specified path.

`.add_possession(new_possession)` Adds another object as a "possession" of the current `Actor` object and establishes a reciprocal "owner" relationship. * **`new_possession`**: The object (e.g., another `Actor`) to be added to the `.Possessions` struct. * **Behavior**: * The `new_possession` object is added as a field to the `.Possessions` struct. * The current `Actor` object is added to the `.Owner` struct of the `new_possession` object. * **Note**: The implementation has a known issue (`FIXME`) where it relies on `inputname()` to name the possession and owner, which may not be reliable in all contexts. A dedicated name property on the `Actor` class is suggested to resolve this. * **Returns**: The updated `Actor` object.

`.drop_possession(possession_name)` Intended to remove a possessed object from the `.Possessions` struct. * **`possession_name`**: A string with the name of the possession to remove. * **Note**: This method is defined but not yet implemented in the current version. * **Returns**: The `Actor` object.

`.add_owner(owner, name_owner)` Adds an owner to the current `Actor` object. This is typically called by the `add_possession` method of the owning object. * **`owner`**: The `Actor` object that will be the owner. * **`name_owner`**: A string representing the name for the owner field in the `.Owner` struct. * **Returns**: The updated `Actor` object.

`.to_table()` Converts the `.Attributes` struct of the `Actor` object into a MATLAB `table`.

A.15 Home\Classes\ActorBuilder.md

ActorBuilder Code

The `ActorBuilder` class implements the builder design pattern to simplify the construction of `Actor` class instances. It separates the complex construction logic from the `Actor` class itself, allowing for a more readable and step-by-step creation process.

An `ActorBuilder` object is configured with the necessary components (like attributes, state, and possessions), and then its `.build()` method is called to generate the final `Actor` object. This approach is particularly useful when an object requires multiple inputs for its creation. Properties and methods are accessed using the dot notation `AB.<Property or Method>` where `AB` is the variable name of the `ActorBuilder` object.

A.15.1 Table of Contents

- Properties
- Methods
    - .read_expanded(fpath)
    - .add_Attributes(Attributes)
    - .add_State(State)
    - .add_Possessions(Possessions)
    - .add_Owner(Owner)
    - .add_Data(Data)
    - .build()

A.15.2 Properties

The `ActorBuilder` class contains the same properties as the `Actor class`.

A.15.3 Methods

This class provides methods for populating its properties and a final method to construct the `Actor` object. To initialize a `ActorBuilder` object use the syntax `AB=ActorBuilder()`.

`.read_expanded(fpath)` This method constructs a partially configured `ActorBuilder` from an expanded actor data format on disk. It reads `attributes.json` and all `.pqt` files from a `State` subdirectory within the specified path. * **`fpath`**: A string representing the file path to the root directory of the expanded actor data path. * **Returns**: A new `ActorBuilder` object with the `Attributes` and `State` properties populated.

`.add_Attributes(Attributes)` This method sets the `Attributes` property for the actor. This typically includes static information like a unique identifier (id), type, or physical characteristics. * **`Attributes`**: A `struct` containing the actor's state data. * **Returns**: The `ActorBuilder` object, allowing for method chaining.

`.add_State(State)` This method sets the `State` property for the actor. Tis can include transient states such as visibility, position, or status that change over time. * **`State`**: A `struct` containing the actor's state data. * **Returns**: The `ActorBuilder` object, allowing for method chaining.

`.add_Possessions(Possessions)` This method sets the `Possessions` property for the actor. It will throw an error if the input is not a `struct`. * **`Possessions`**: A `struct` containing the actor's possessions. * **Returns**: The `ActorBuilder` object, allowing for method chaining.

`.add_Owner(Owner)` This method sets the `Owner` property for the actor. * **`Owner`**: A `struct` containing the actor's owner. * **Returns**: The `ActorBuilder` object, allowing for method chaining.

`.add_Data(Data)` This method sets the `Data` property for the actor. It will throw an error if the input is not a `TimeSeriesCollection` object. * **`Data`**: A `TimeSeriesCollection` object. * **Returns**: The `ActorBuilder` object, allowing for method chaining.

`.build()` This method constructs and returns the final `Actor` object using all the properties previously configured on the builder.

A.16 Home\Classes\ActorCollection.md

ActorCollection Code

The `ActorCollection` class is a container designed to store and manage multiple `Actor` objects. It provides a high-level interface for performing batch operations, such as filtering a subset of actors based on specific criteria and generating descriptive statistics across the entire collection.

This class simplifies the analysis of multi-actor scenarios by aggregating actors and offering methods to query and summarize their attributes and states. Properties and methods are accessed using the dot notation `AC.<Property or Method>;` where `AC` is the variable name of the `ActorCollection` object.

A.16.1 Table of Contents

- Properties
- Methods
    - .read_expanded(fpath)
    - .describe()
    - .describe_Attributes()
    - .describe_State()
    - .filter(logic_fn)
    - .filter_by_attributes(field_name, logic_fn)
    - .add(actor)
    - .drop(idx_drop)
    - .to_pqt(fpath)

A.16.2 Properties

The `ActorCollection` class contains the following property:

- **`Collection`**: An array that stores the individual `Actor` objects.

A.16.3 Methods

This class provides methods for creating, modifying, and analyzing the collection of actors.

`.read_expanded(fpath)` This method reads a directory containing multiple actor data sets, where each sub-directory represents a single actor in the expanded format. It iterates through the sub-directories, creates an `Actor` object for each one using `Actor.read_expanded`, and appends a `ActorCollection` with them. * **`fpath`**: A string representing the file path to the parent directory containing the individual actor folders. * **Returns**: A new `ActorCollection` object containing all actors found in the specified path.

`.describe()` This method generates a high-level summary of the entire collection by analyzing the `Attributes` and `State` data across all actors. It serves as a wrapper for `.describe_Attributes()` and `.describe_State()`. * **Returns**: A `struct` with two fields: `Attributes` and `State`, each containing a table of descriptive statistics.

`.describe_Attributes()` This method computes descriptive statistics of the `Attributes` for all actors in the collection. It identifies all unique attribute names and, for each one, computes statistics such as the number of actors possessing it, the data type, and a summary of its values. For numeric attributes, it calculates mean, standard deviation, and median. For categorical attributes, it lists

the unique values and their frequencies. * **Returns**: A `table` summarizing the attributes across the collection.

`.describe_State()` This method computes descriptive statistics of the `State` data across all actors in the collection. For each unique state variable found, it calculates the number of actors that have that state, the average number of state events per actor, and the average duration of those events. * **Returns**: A `table` summarizing the state data across the collection.

`.filter(logic_fn)` This method filters the collection to create a new `ActorCollection` containing only the actors that satisfy a specified condition. The condition is defined by a function handle that returns `true` for actors that should be kept. * **`logic_fn`**: A function handle that takes an `Actor` object as input and returns a logical value (`true` or `false`). Example: `@(x) x.Attributes.role == "Soldier"`. * **Returns**: A new, filtered `ActorCollection` object.

`.filter_by_attributes(field_name, logic_fn)` This method provides a more direct way to filter actors based on the value of a specific attribute. * **`field_name`**: A string representing the name of the attribute field to filter on. * **`logic_fn`**: A function handle that takes the attribute's value as input and returns a logical value. Example: `@(x) contains(x, 'BadGuy')`. * **Returns**: A new, filtered `ActorCollection` object.

`.add(actor)` This method adds a new `Actor` object to the end of the collection. * **`actor`**: The `Actor` object to be added to the collection. * **Returns**: An updated `ActorCollection` object with the new actor included.

`.drop(idx_drop)` This method removes one or more actors from the collection based on their numerical index. * **`idx_drop`**: An integer or an array of integers specifying the index/indices of the actor(s) to remove. * **Returns**: An updated `ActorCollection` object with the specified actors removed.

`.to_pqt(fpath)` This method serializes the entire `ActorCollection` into a single Parquet file. It first converts the collection into a structured table and then writes it to the specified file path. * **`fpath`**: A string representing the full file path (including name and `.pqt` extension) where the output file will be saved. GazeOriginSeries Code

A.17 Home\Classes\GazeOriginSeries.md

The `GazeOriginSeries` class is a subclass of `TimeSeries` designed specifically for handling STAIT-formatted time-series data representing an actor's gaze.

As a subclass, it inherits all the properties and methods of the parent `TimeSeries` class. It extends this functionality by providing validation methods to ensure the data complies with the specific requirements for gaze data, such as the presence of location and direction vector components. Properties and methods are accessed using dot notation such as `GD.<Property or Method>` where GD is the variable name of the GazeOriginSeries object.

A.17.1 Table of Contents

- Properties
- Methods
  - .read_stait(fpath)
  - .check_standard()
  - .check_data()

A.17.2 Properties

The `GazeOriginSeries` inherits properties directly from the parent `TimeSeries` class.

A.17.3 Methods

This class uses all methods inherited from `TimeSeries` and provides its own specialized methods.

`.read_stait(fpath)` This method reads a STAIT-formatted `.pqt` file and its associated `.json` metadata file. It then uses the `TimeSeriesBuilder` to construct and return an object specifically of the `GazeOriginSeries` class, ensuring the correct object type is created from the start. * **`fpath`**: A string representing the file path and name, without the extension. * **Returns**: A new `GazeOriginSeries` object.

`.check_standard()` This method verifies that the `.Data` table contains all the required columns: `time`, `location_x`, `location_y`, `location_z`, `direction_x`, `direction_y`, and `direction_z`. * **Returns**: Returns `true` if the object's data conforms to the standard. It will throw an error if any required fields are missing.

`.check_data()` This method checks for the same set of required fields as `.check_standard()`. * **Behavior**: If all required fields are present in the `.Data` table, it will display a confirmation message to the console. If any fields are missing, it will throw an error that lists the specific fields that are absent. TimeSeries Code

A.18 Home\Classes\TrackerSeries.md

TrackerSeries Code

The `TrackerSeries` class is a subclass of `TimeSeries` designed for handling STAIT-formatted time-series data representing the position and orientation of an actor in world coordinates.

As a subclass, it inherits all the properties and methods of the parent `TimeSeries` class. It extends this functionality by providing validation methods to ensure the data complies with the specific requirements for tracker data, such as the presence of location and orientation vectors. Properties and methods are accessed using the dot notation `TD.<Property or Method>` where TD is the variable name of the gaze data object in STAIT format.

A.18.1 Table of Contents

- Properties
- Methods
  - .read_stait(fpath)
  - .check_standard()
  - .check_data()

A.18.2 Properties

The `TrackerSeries` inherits properties directly from the parent `TimeSeries` class.

A.18.3 Methods

This class uses all methods inherited from `TimeSeries` and provides its own specialized methods.

`.read_stait(fpath)` This method reads a STAIT-formatted `.pqt` file and its associated `.json` metadata file. It then uses the `TimeSeriesBuilder` to construct and return an object specifically of the `TrackerSeries` class, ensuring the correct object type is created from the start. * **`fpath`**: A string representing the file path and name, without the extension. * **Returns**: A new `TrackerSeries` object.

`.check_standard()` This method verifies that the `.Data` table contains all the required columns for position (`location_x`, `location_y`, `location_z`) and orientation vectors (`forward_x/y/z`, `left_x/y/z`, `up_x/y/z`), in addition to `time`. * **Returns**: Returns `true` if the object's data conforms to the standard. It will throw an error if any required fields are missing.

`.check_data()` This method provides another layer of validation for the tracker data format. It checks for the same set of required fields as `.check_standard()`. * **Behavior**: If all required fields are present in the `.Data` table, it will display a confirmation message to the console. If any fields are missing, it will throw an error that lists the specific fields that are absent. Events!

A.19 Home\Functions.md

+process [Link!] Process is the name space for functions that modify and manipulate data such as filtering or transforming data into other spaces. Use `process.<function>` syntax to apply process functions.

+detect [Link!] Detect is the name space for functions that are used to detect events such as saccades or fixations. Use `detect.<function>` syntax to apply detect functions.

+visualize [Link!] Visualize is the name space for functions used to visualize data – coming soon!

A.20 Home\Functions\+detect.md

+detect Code

Tutorial script

The `+detect` package provides functions for identifying specific events within `TimeSeries` data.

Syntax: `detect.<function_name>(ts, ...)`

A.20.1 Table of Contents

- Functions
  - .gaze_blink(ts, min_duration, max_duration)
  - .gaze_fixation(ts, …)
  - .gaze_saccade(ts, …)

A.20.2 Functions

The `+detect` package contains the following functions for event detection.

`.gaze_blink(ts, min_duration, max_duration)` This function detects blinks by identifying periods of missing gaze data within a `GazeOriginSeries` object. * **`ts`**: A `GazeOriginSeries` object. * **`min_duration`** (optional): The minimum duration in milliseconds (ms) for an event to be considered a blink.

Events shorter than this are discarded. Default: `10`. * **`max_duration`** (optional): The maximum duration in milliseconds (ms) for an event to be considered a blink. Events longer than this are discarded. Default: `3000`. * **Returns**: A `table` with `start_time`, `stop_time`, and `type` for all detected blinks.

`.gaze_fixation(ts, min_duration, max_duration, min_velocity, max_velocity, threshold, median_filt_order, sg_filt_length)` This function detects fixations by applying an adaptive velocity threshold algorithm to the gaze direction data. It identifies periods of relative stability where the angular velocity of the gaze remains below a defined threshold. * **`ts`**: A `GazeOriginSeries` object. * **`min_duration`**: Minimum duration (in ms) for a fixation. Detected fixations below this threshold are removed. Default: `20`. * **`max_duration`**: Maximum duration (in ms) for a fixation. Detected fixations above this threshold are removed. Default: `300`. * **`min_velocity`**: Minimum velocity threshold (in deg/s). After detection, fixations with a maximum velocity below this are removed. Default: `50`. * **`max_velocity`**: Maximum velocity threshold (in deg/s). After detection, fixations with a maximum velocity above this are removed. Default: `700`. * **`threshold`**: The initial velocity threshold (in deg/s) for the adaptive detection algorithm. Default: `50`. * **`median_filt_order`**: The order for an optional median filter applied to the gaze direction data before velocity calculation. If empty or not provided, the filter is not applied. Default: `[]`. * **`sg_filt_length`**: The length (in samples) of an optional second-order Savitzky-Golay filter applied to the gaze velocity data to smooth peaks. If empty or not provided, the filter is not applied. Default: `[]`. * **Returns**: A `table` with `start_time`, `stop_time`, and `type` for all detected and valid fixations.

`.gaze_saccade(ts, min_duration, max_duration, min_velocity, max_velocity, threshold, median_filt_order, sg_filt_length)` This function detects saccades by applying an adaptive velocity threshold algorithm to the gaze direction data. It identifies rapid eye movements where the angular velocity exceeds a defined threshold. * **`ts`**: A `GazeOriginSeries` object. * **`min_duration`**: Minimum duration (in ms) for a saccade. Detected saccades below this threshold are removed. Default: `20`. * **`max_duration`**: Maximum duration (in ms) for a saccade. Detected saccades above this threshold are removed. Default: `300`. * **`min_velocity`**: Minimum peak velocity threshold (in deg/s). After detection, saccades with a peak velocity below this are removed. Default: `50`. * **`max_velocity`**: Maximum peak velocity threshold (in deg/s). After detection, saccades with a peak velocity above this are removed. Default: `700`. * **`threshold`**: The initial velocity threshold (in deg/s) for the adaptive detection algorithm. Default: `50`. * **`median_filt_order`**: The order for an optional median filter applied to the gaze direction data before velocity calculation. If empty or not provided, the

filter is not applied. Default: `[]`. * **`sg_filt_length`**: The length (in samples) of an optional second-order Savitzky-Golay filter applied to the gaze velocity data to smooth peaks. If empty or not provided, the filter is not applied. Default: `[]`. * **Returns**: A `table` containing `start_time`, `stop_time`, `type`, `duration_ms`, `length_deg`, `max_velocity`, `mean_velocity`, and `median_velocity` for all detected and valid saccades.

A.21 Home\Functions\+process.md

+process Code

Tutorial code

The `+process` package is a collection of MATLAB functions designed for processing `TimeSeries` objects and their subclasses. These functions provide common operations such as filtering, resampling, and coordinate transformations.

Each function is designed to be used either directly or through the `TimeSeries.pipe()` method, which allows for chaining multiple processing steps in a clean and readable manner.

Syntax: `process.<function_name>(ts, ...)` **Pipe Syntax:** `ts.pipe(@process.<function_name>, ...)`

A.21.1 Table of Contents

- Functions
  - .add_azieli(ts)
  - .add_euler(ts, rot_order)
  - .add_quat(ts)
  - .derivative(ts, vargin)
  - .median_filt(ts, filt_ord, vargin)
  - .resample(ts, new_rate,vargin)
  - .resample_by_timestamp(ts, new_timestamps,vargin)

A.21.2 Functions

The `+process` package contains the following functions for data manipulation and analysis.

`.add_azieli(ts)` This function converts 3D direction vector data into a 2D horizontal coordinate system (azimuth and elevation). This can be useful when comparing head tracking with gaze or in cases where roll of an object is minimal.

* **ts**: A TrackerSeries or GazeOriginSeries object. * **Returns**: A new object of the same class with azimuth and elevation columns added.

.add_euler(ts, rot_order) This function converts orientation data from a three-vector representation (forward, left, up) into a set of three Euler angles (roll, pitch, yaw). * **ts**: A TrackerSeries object. * **rot_order** (optional): A string specifying the rotation order. If not provided, it defaults to 'XYZ'. Valid options are: * 'XYZ': Roll (X), Pitch (Y), Yaw (Z) * 'ZYX': Yaw (Z), Pitch (Y), Roll (X) * 'YXZ': Pitch (Y), Roll (X), Yaw (Z) * 'XZY': Roll (X), Yaw (Z), Pitch (Y) * 'ZXY': Yaw (Z), Roll (X), Pitch (Y) * **Returns**: A new TrackerSeries object with Euler angle columns added.

.add_quat(ts) This function converts orientation data from the three-vector representation (forward, left, up) to a scalar-first (w, x, y, z) quaternion representation. * **ts**: A TrackerSeries object. * **Returns**: A new TrackerSeries object with quaternion columns added.

.derivative(ts, ...) This function applies a first-order time derivative to one or more data columns. By default, it processes all numeric columns except for time. * **ts**: Any TimeSeries object. * **vargin** (optional): Name-value pairs. Use 'ColumnsToProcess', {'col1', 'col2'} to specify which columns to process. * **Returns**: A new TimeSeries object where the specified columns contain the time-derivative of the original data.

.median_filt(ts, filt_ord, vargin) This function applies a 1D median filter to one or more data columns. By default, it filters all numeric columns except for time. * **ts**: Any TimeSeries object. * **filt_ord**: An integer defining the filter length (order). * **vargin** (optional): Name-value pairs. Use 'ColumnsToProcess', {'col1', 'col2'} to specify which columns to filter. * **Returns**: A new TimeSeries object where the specified columns contain the filtered data.

.resample(ts, new_rate, vargin) This function resamples the time-series data to a new, constant sampling rate. It uses MATLAB's resample function, which includes an anti-aliasing filter to prevent distortion. This operation affects all data columns and the time vector. * **ts**: Any TimeSeries object. * **new_rate**: The target sampling rate in Hertz (Hz). * **vargin** (optional): Name-value pairs excluding ColumnsToPorcess. * **Returns**: A new TimeSeries object containing the resampled data.

.resample_by_timestamp(ts, new_timestamps, vargin) This function resamples the time-series data to a new set of specified timestamps. It uses MATLAB's resample function to perform the interpolation. This operation affects all data columns and the time vector. * **ts**: Any TimeSeries object. *

`new_timestamps`: A vector of timestamps to which the data should be resampled. * **`vargin`** (optional): Name-value pairs excluding `ColumnsToPorcess`. * **Returns**: A new `TimeSeries` object containing the resampled data. Example script. [Link]

Syntax: `visualize.<function>`

Coming soon to a repo near you! Functions!

A.22 Tutorials.md

A.22.1 Wrangling Tutorial

A.22.1.1 Analysis Tutorial Code

How to import in STAIT data to MatLab

How to use +detect namespace functions - specifically saccade detection

How to use +process namespace functions

How to create custom analysis functions

How to use actor class and methods

A.22.2 GIT

GIT 101

GIT commands quick reference

A.22.3 Calibrating Coordinates in XR Space

We have developed a standard procedure for matching sensor coordinates to real-world coordinates.XR_StandardMovements.docx

A.23 Tutorials\Wrangling.md

“Data that is loved tends to survive.” — Kurt Bollacker

Wrangling will be specific to your particular devices, experiment and data collection pipeline but at a minimum must be able to satisfy data requirements. Development of our wrangling code is done using Python.

A.24 Examples

1. Unreal/UETack + Meta Quest Pro + Noraxon IMUs
2. PSI/SimpleRecorder + HoloLens2 + Dyad

A.24.1 Wrangling Overview

What's needed of the STAIT wrangling process:

- Love for and patience with your data.
- Knowledge of device timing and units to align time stamps.
- Knowledge of the coordinate systems and spatial origins of your raw data.
- Knowledge on the operations and code to transform those coordinate systems to the STAIT standard.
- Recording of required metadata and creation of metadata during wrangling process such as attributes.

Our wrangling process consists of: 1) Raw conversion to Level 0 files 2) Level 0 file conversion to Level 1

Raw to Level 0 steps: 1. Set temporal origin by finding earliest time point across raw data streams and set as time 0. Convert time into floats in milliseconds. 2. Transform and/or rotate spatial coordinates into STAIT standard. Standardized origin is world coordinates in cm. 3. Here properties are assigned using metadata and standardized rules. 4. Any non-standardized or non-essential information is excluded from STAIT level 0 data. [1]
5. We have set a rule such that any manipulation of time-series data MUST be invertible. This means no: Non-linear transforms, filters, interpolation, resampling, replacement of null values, etc. 6. Creation of properties including metadata, IsMissing table, IsAnamoulous table, and DataAttributes table. DataAttributes include, Units, Manifold and Domain. 7. Exporting of metadata to .json and spatiotemporal data to .pqt files.

Level 0 file conversion level 1 steps: 1. Convert time into integers (reflecting ms). 2. Align all data to start/stop time such that row data time matches across all level 1 files. 3. Linear interpolation is used to upsample/downsample all data to common sampling rate which is currently set at 1000Hz. Data points with missing data before and after time periods that are 2x the original sampling rate are removed from interpolation. 4. Exporting of data.

---

[1] While you can still wrangle and convert non-standardized raw spatiotemporal data into a TimeSeries object (e.g. pupil size data), not all functionality will be applicable. For example, if the STAIT standardized metadata requirements is attached to pupil size data, resampling or slicing the pupil data using built-in STAIT functinos is still possible.

A.25 VR Data Wrangling Example

[Associated Data location] (the LINK)
Example Script raw to level 00
Example Script level 00 to level 01

This example outlines a procedure for wrangling raw virtual reality (VR) experiment data into the standardized STAIT data levels. The primary tools are a set of Python scripts designed to process data from the UETack data collection pipeline, the Meta Quest Pro with controllers, and Noraxon IMUs.

A.25.1 Process Overview

The wrangling process is divided into two main stages:

1. **Raw to Level 00:** This initial step involves aggregating raw data files, synchronizing time across different recording devices, transforming data into a standardized coordinate system, defining attributes and metadata, and saving the results. This stage is handled by `STRIKE_raw_to_00.py`.
2. **Level 00 to Level 01:** This second step takes the processed Level 00 data and standardizes its temporal resolution by resampling all data streams to a uniform frequency. This stage is handled by `STRIKE_00_to_01.py`.

A.25.2 Step 1: Raw to Level 00

The `STRIKE_raw_to_00.py` script executes a series of functions to convert the raw, multi-folder output from a data collection session into a unified Level 00 dataset for a single subject. This script calls Actors.py, uetack.py, meta_quest.py, general.py and noraxon.py.

A.25.2.1 File Aggregation

Our UETack data collection pipeline creates a new output folder for each run (every start and stop of Tack within each session). The first wrangling step is to traverse these folders and combine the data from corresponding files (e.g., all `camera_positions.tsv` files are merged into one).

- **Implementation:** The `uetack.load_client_data()` function reads all subdirectories within a subject's `dump` folder, finds the specified file in each, and concatenates them into a single Pandas DataFrame.

A.25.2.2 Time Synchronization

To create a unified temporal reference, we must align timestamps from different sources (UETack's UTC, system clock, and Noraxon's internal clock).

1. **Time Map Generation:** The `uetack.load_time_map()` function reads all `time.tsv` files from the UETack dumps. It combines them and creates a master mapping between the `utc` timestamps, the system's `posix_time`, and a continuous `server_time` that spans the entire collection session.
2. **Server Time Conversion:** For each data stream that uses UTC timestamps (e.g., head, controllers, gaze), the `uetack.utc_to_servertime()` function uses this map to interpolate and assign a `server_time` to each data point.
3. **IMU Synchronization (Noraxon):** For Noraxon IMU data, which runs on a separate clock, the `noraxon.sync_imu_xdf()` function aligns the IMU's analog sync pulse with the digital sync pulse recorded in an `.xdf` file from the main system. This allows us to calculate the correct `server_time` for each IMU sample.

A.25.2.3 Data Stream Wrangling and Coordinate Transformation

Each data stream is loaded, transformed into the standardized STAIT coordinate system and has attributes defined.

HEAD TRACKING (`VR_HEAD`)

- **Source:** `camera_positions.tsv`
- **Function:** `meta_quest.wrangle_head()`
- **Transformations:**
  - Y axis Location data is inverted: $y_{STAIT} = -y_{raw}$.
  - Orientation, originally stored as a quaternion, is converted into three orthogonal vectors: `forward`, `up`, and `left`. This is done by applying the quaternion's rotation to basis vectors. The Y-components of the resulting vectors are also inverted.

GAZE TRACKING (`VR_GAZE_MAIN`, `VR_GAZE_LEFT`, `VR_GAZE_RIGHT`)

- **Source:** `traces_main.tsv`, `traces_left.tsv`, `traces_right.tsv`
- **Function:** `meta_quest.wrangle_gaze()`
- **Transformations:**
  - Gaze location (`trace_start`) is inverted: $ y_{STAIT} = -y_{raw} $.
  - A normalized gaze `direction` vector is calculated using the gaze start and impact points. The Y-component of this vector is also inverted.

CONTROLLERS (`VR_CONTROLLER_LEFT`, `VR_CONTROLLER_RIGHT`)

- **Source:** `client_xr_motion_controller.tsv`, `pawn_positions.tsv`

- **Function:** `meta_quest.wrangle_controllers()`
- **Transformations:**
    - Controller data is recorded relative to the player's "pawn" in the VR environment. To get world coordinates, the pawn's position and orientation data are resampled to match the controller's timestamps.
    - Controller world location is calculated by applying offsets to the pawn's position.
    - Controller world orientation is calculated by applying the controller's rotation to the pawn's orientation vectors.
    - The Y-axis is inverted for final location and orientation vectors.

ACTORS (`ACTORS/`)

- **Source:** `actors.json`, `actor_lifetimes.tsv`
- **Function:** `uetack.load_actors_dump()`, `uetack.actors_dump_to_ActorCollection()`
- **Processing:**
    - Information about all actors (objects, characters, etc.) that appeared in the environment is loaded.
    - The `actor_lifetimes.tsv` file is used to determine when each actor was "destroyed" or removed from the scene.
    - This information is converted into a `stait.structures.ActorCollection` object, which provides a structured representation of each actor's attributes (e.g., spawn location) and state over time (e.g., spawned, despawned).
    - Coordinate transformations (Y-axis inversion, converting right vector to left vector) are applied to the spawn orientation.

A.25.2.4 Temporal Epoch Alignment

After all data streams are loaded and time-synchronized, the `uetack.align_epoch()` function finds the earliest `server_time` across all datasets. This timestamp is treated as the temporal origin (

$$t = 0$$

) for the entire experiment. This origin time is subtracted from all time columns, and the original start time (in UTC) is stored in the metadata as `Time Epoch`.

A.25.2.5 Output Generation

All processed DataFrames are saved to the subject's Level 00 directory.

- **Data Streams:** Saved as Apache Parquet files (`.pqt`) (e.g., `vr_head.pqt`). This format includes the data and its associated metadata which is output as JSON files (attributes, sample rate, etc.).
- **Actors:** The `ActorCollection` is written to a dedicated `Actors` folder using the `write_expanded` method.
- **Time Map:** The master `time_map` is saved as `time_map.pqt` for reference and debugging.

A.25.3 Step 2: Level 00 to Level 01

The purpose of this step is to create a dataset where all time-series data share a common, uniform sampling rate. This is essential for many time-series analysis techniques and for simplifying data fusion.

- **Script:** `STRIKE_00_to_01.py`
- **Function:** `stait.wrangling.general.level00_to_level01_batch()`
- **Process:**

  1. The script iterates through all Level 00 `.pqt` files for a subject.
  2. Each DataFrame is resampled to a fixed frequency (e.g., 100 Hz as defined by `resample_rate`). Linear interpolation is used to fill the data.
  3. The resampled DataFrames are saved as new `.pqt` files in the corresponding subject's `level01` directory.

A.26 AR Data Wrangling Example

[Associated Data location] (the LINK)
Example Script raw to level 00
Example Script level 00 to level 01

This example shows python scripts for wrangling data from two simultaneous participants donning a HoloLens2 recorded via PSI using the custom Simple Recorder software.

A.26.1 Process Overview

1. **Raw to Level 00:** This step processes the raw text files for all dyadic runs. It synchronizes participant timelines within each run to a common starting point and saves the data in a structured format. This stage is handled by `R2C2_raw_to_00_batch.py`.
2. **Level 00 to Level 01:** This step takes the processed Level 00 data and resamples all time-series streams to a uniform frequency across each dyad, creating a single, shared timeline for each run. This stage is handled by `R2C2_00_to_01_batch.py`.

A.26.2 Step 1: Raw to Level 00

A.26.2.1 Data Loading and Parsing

For each run, the raw data is organized into subdirectories for each participant, containing an `export` folder with the data streams.

- **Implementation:** The `hololens2.wrangle_dyad()` function is called for each run. It reads the `.txt` file for each data stream (e.g., `Head/Head.txt`) within each participant's folder.
- **Parsing:** Each stream has a specific `wrangle_*` function (e.g., `wrangle_pose`, `wrangle_eyes`) that parses the tab-delimited text file, assigns correct column names, and selects the relevant data columns.

A.26.2.2 Time Synchronization and Epoch Alignment

HoloLens 2 timestamps are recorded as 100-nanosecond intervals since January 1, 0001. The wrangling process converts this to a more usable format and aligns the data from both participants within a run.

1. **Timestamp Conversion:** For each data point, the raw timestamp is converted to milliseconds.
2. **Dyadic Epoch Alignment:** The `hololens2.wrangle_dyad()` function finds the earliest timestamp across **all data files from both participants within a single run**. This global minimum time is designated as that run's temporal origin. This value is then subtracted from all timestamps in the dyad, ensuring all data for that run begins on a common timeline.
3. **Epoch Metadata:** The original start time is converted to a standard UTC timestamp and stored in the metadata of each file as `Time Epoch`.

A.26.2.3 Data Stream Overview

The script processes the following primary data streams:

- **Head** (`head`):
  - **Source:** `Head/Head.txt`
  - **Content:** Provides the 6 degrees of freedom (6DOF) pose (location and orientation) of the user's head in meters. Orientation is given as `forward`, `up`, and `left` basis vectors.
- **Gaze** (`gaze`):
  - **Source:** `Eyes/Eyes.txt`
  - **Content:** Provides the origin (location) and `direction` vector of the user's combined eye gaze.
- **Hands** (`hands_left`, `hands_right`):
  - **Source:** `Hands/Left.txt`, `Hands/Right.txt`
  - **Content:** Provides detailed tracking for each hand, including a boolean for whether the hand is active and the pose for 26 individual joints.
- **Video Pose** (`video_pose`):
  - **Source:** `Video/Pose.txt`
  - **Content:** Provides the 6-DOF pose of the world-facing camera for each video frame.
- **Video Timing** (`video_timing`):
  - **Source:** `Video/Timing.txt`
  - **Content:** Maps each video `frame_number` to its corresponding timestamp.

A.26.2.4 Output Generation

The processed DataFrames for each participant are saved into structured Level 00 directories (`level00/<run_name>/<participant_name>/`) as Apache Parquet files (`.pqt`) and JSON files containing metadata.

A.26.3 Step 2: Level 00 to Level 01

The `R2C2_00_to_01_batch.py` script creates a dataset where all time-series data from both participants in a run share a common, uniform sampling rate.

- **Function:** `hololens2.level00_to_level01_dyad()`

- **Process:**
    1. The script iterates through each run in the Level 00 directory. For each run, it loads all `.pqt` files for **both** subjects in the dyad.
    2. It determines a global timeline for the run by finding the start time ($ t=0 $) and the very last timestamp (`time_stop`) across all loaded files for that dyad.
    3. A new, uniform time vector is generated from `time_start` to `time_stop` at the specified `resample_rate` (e.g., $ 1000 $).
    4. The data for every channel is resampled onto this new time vector using linear interpolation (`np.interp`).
    5. **Missing Data Labeling:** To preserve data integrity, the script identifies time gaps in the original Level 00 data. For each resampled column, a corresponding boolean column (`is_missing_*`) is added. This flag is set to `True` for any timestamp that falls within a gap in the original data, making it easy to distinguish between true and interpolated data points.
    6. **Frustum Creation:** A new `frustrum` data stream is created by combining the resampled `video_pose` and `video_timing` data. This stream provides the camera's pose and corresponding frame number on the unified Level 01 timeline.
- **Output:** The fully resampled DataFrames are saved as new `.pqt` files in the corresponding Level 01 directory structure (`level01/<run_name>/<participant_name>/`).

List of Symbols, Abbreviations, and Acronyms

3D	Three-Dimensional
6DOF	6 Degrees of Freedom
AR	Augmented Reality
BIDS	Brain Imaging Data Structure
GPS	Global Positioning System
IMU	Inertial Measurement Unit
MR	Mixed Reality
ROS	Robot Operating System
STAIT	Spatiotemporal Analysis and Inference Toolkit
VR	Virtual Reality

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CB TECH LIB