dv-processing

Release rel_1_7

iniVation AG

Feb 08, 2024

GETTING STARTED

Generic processing algorithms for event cameras.

CHAPTER

ONE

INTRODUCTION

This documentation describes the API and the algorithms available in the `dv-processing` library. The documentation covers the basic usage of the library for event camera data processing. The library builds on top of C++20 coding standard, provides state-of-the-art algorithms to process event data streams from iniVation cameras with high efficiency. The library also provides Python bindings that allows users to develop high performance event processing application in Python as well. Since the library is built on top of modern C++, it extensively uses template metaprogramming to provide extensible and performant implementations of event processing algorithms. Python bindings have some limitations due to the of use templates, but most applications can be developed using Python alone. Extensive code samples in both C++ and Python are provided next to algorithm description, the samples also follow modern coding style conventions. More information about the coding style convention can be found here[1]. Code samples apply the constant and immutability rules[2] from the previously mentioned document.

[1] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
[2] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#S-const

CHAPTER

TWO

GETTING STARTED

This section covers installation and usage of the library in Linux , Windows , and MacOS . Usage in CMake projects is covered for the C++ API, and pip installation is preferred for Python projects.

2.1 Installation of dv-processing

The dv-processing library can be installed using different package management tools as well as installations from source. Since it is a header-only library, it can be used as a submodule in other CMake C++ projects.

2.1.1 C++

Installation of dv-processing headers is available on Linux, MacOS, and Windows platforms.

Linux

Ubuntu 18.04 (apt)

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo add-apt-repository ppa:inivation-ppa/inivation-bionic
sudo apt-get update
sudo apt-get install dv-processing
```

Ubuntu 20.04 / 22.04 (apt)

```
sudo add-apt-repository ppa:inivation-ppa/inivation
sudo apt-get update
sudo apt-get install dv-processing
```

Fedora Linux

```
sudo dnf copr enable inivation/inivation
sudo dnf install dv-processing
```

Arch Linux

You can find dv-processing in the AUR repository, install the package 'dv-processing'.

The standard packages in the AUR repository already include all development files.

Gentoo Linux

A valid Gentoo ebuild repository is available here[3] over Git. The package to install is 'dev-libs/dv-processing'.

The standard packages in the Gentoo ebuild repository already include all development files.

MacOS (brew)

Please notice that MacOS installation requires a recent compiler version (Apple XCode >= 14.3 or LLVM >= 13).

```
brew tap inivation/inivation
brew install libcaer --with-libserialport --with-opencv
brew install dv-processing
```

Windows (VCPKG)

Windows installation is supported using the VCPKG package manager. This requires cloning of the VCPKG repository, you can follow the official quick start guide to get started[4]. The library can be installed using `.\vcpkg.exe install dv-processing` after VCPKG has been downloaded and bootstrapped.

A short tutorial for installation, create a directory at `C:\src`, execute these commands there:

```
git clone https://github.com/microsoft/vcpkg
.\vcpkg\bootstrap-vcpkg.bat
.\vcpkg\vcpkg install dv-processing
```

Installing `dv-processing[tools]` will also compile and install the command-line utilities, such as dv-filestat.

[3] https://gitlab.com/inivation/gentoo-inivation/
[4] https://github.com/microsoft/vcpkg#quick-start-windows

Installing as a git submodule

The library can be used in a CMake project by adding it as a git submodule. To add the library using git, call the following command from your project directory:

```
git submodule https://gitlab.com/inivation/dv/dv-processing.git thirdparty/dv-
↪processing
```

This will add the source code of the latest released version of dv-processing to a directory `thirdparty/dv-processing`. Now you can enable it in your CMakeLists.txt:

```
ADD_SUBDIRECTORY(thirdparty/dv-processing EXCLUDE_FROM_ALL)

# link your targets against the library
TARGET_LINK_LIBRARIES(your_target
        dv::processing
        ...)
```

Build and install from source

Manual build and installation from source is also possible. The library requires a C++20 compatible compiler and installed dependencies.

Please follow the installation instructions available in the source code repository[5] to install the library from source.

2.1.2 Python

The dv-processing library is also available in Python using Pybind11 bindings. Since the underlying implementation remains C++ and only exposes the API through Python bindings, the performance of the provided methods remains high.

Installation using `pip install` is recommended, as it works on all supported operating systems, as well as VirtualEnv and Conda environments. On Linux systems, installation can also be performed through the package manager for the system Python installation.

Pip

```
python3 -m pip install dv-processing
```

Note: Pip installation is the only supported method for installation in a virtual Python environment (VirtualEnv / Conda).

[5] https://gitlab.com/inivation/dv/dv-processing/

Linux

Ubuntu 18.04 (apt)

```
sudo add-apt-repository ppa:ubuntu-toolchain-r/test
sudo add-apt-repository ppa:inivation-ppa/inivation-bionic
sudo apt-get update
sudo apt-get install dv-processing-python
```

Ubuntu 20.04 / 22.04 (apt)

```
sudo add-apt-repository ppa:inivation-ppa/inivation
sudo apt-get update
sudo apt-get install dv-processing-python
```

Fedora Linux

```
sudo dnf copr enable inivation/inivation
sudo dnf install dv-processing-python
```

Arch Linux

You can find dv-processing in the AUR repository, install the package 'dv-processing', which includes the Python bindings.

Gentoo Linux

A valid Gentoo ebuild repository is available here[6] over Git. The package to install is 'dev-libs/dv-processing' with the USE flag 'python' enabled.

2.2 Usage in a project

The library can be used in C++ CMake projects after a successful installation. Python projects support conda and venv environments using pip installation, pip installation is the preferred method for use in python, although system-wide installation packages are also provided for some linux distributions.

[6] https://gitlab.com/inivation/gentoo-inivation/

2.2.1 C++ (CMake)

To use the installed version of dv-processing in your project, add these lines to your CMakeLists.txt:

```
# Find installed dv-processing.
FIND_PACKAGE(dv-processing)

# Link your targets against the library
TARGET_LINK_LIBRARIES(your_target
        dv::processing
        ...)
```

2.2.2 Python

After installing the dv-processing python bindings either using pip or system-wide installation, the library is included using a simple import statement:

```
import dv_processing as dv

print(dv.__version__)
```

Note: Pip installation is the only supported method for installation in a virtual python environment (conda / venv).

CHAPTER

THREE

BASICS

This section covers the very basic features of this library:

- Storing events in memory and efficiently accessing them;
- Frame image accumulation algorithms from events;
- Noise filtering in events and efficient subsampling;
- Input/Output of event data: live camera access, as well as reading / writing data from a file.

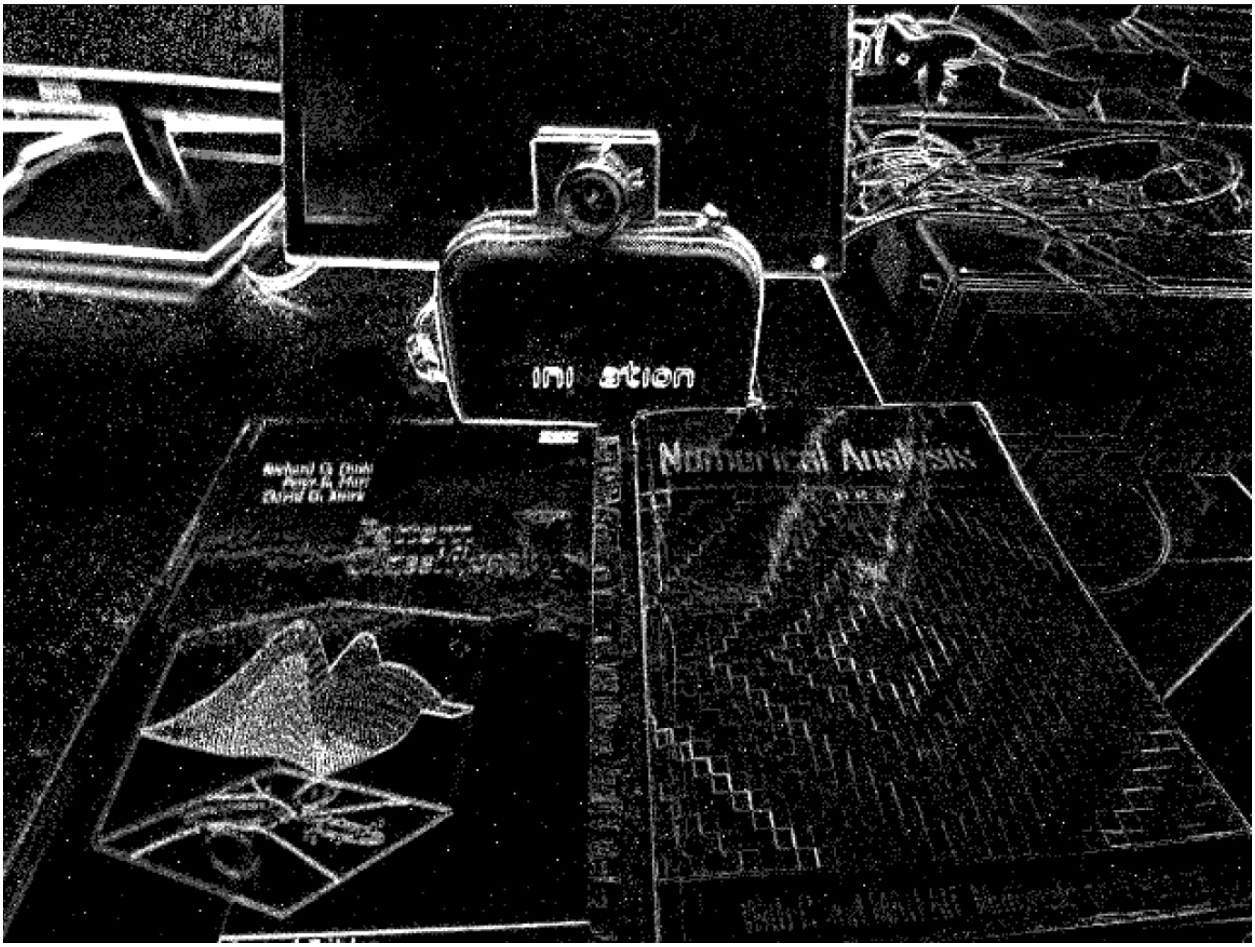


Fig. 1: Frame generated using `dv::EdgeMapAccumulator` class.

3.1 Event store

The dv-processing library provides an easy to use and an efficient data structure to store and manage incoming event data from the camera - the *dv::EventStore* class. It is implemented as a shallow, shared ownership data structure which only holds pointers to actual memory locations containing event data. Similarly to how OpenCV handles image data, the *dv::EventStore* does not copy the data, instead it only holds pointers to received packets of events and uses metadata of these packets to efficiently slice the data given time-intervals.

3.1.1 Create and store events

The following sample code shows how to create an empty event store and fill it with software generated events:

C++

Python

```cpp
#include <dv-processing/core/core.hpp>
#include <dv-processing/core/utils.hpp>

int main() {
    // Initialize an empty store
    dv::EventStore store;

    // Get the current timestamp
    const int64_t timestamp = dv::now();

    // Add some events into the event store
    // This allocates and inserts events at the back, the function arguments are:
    // timestamp, x, y, polarity
    store.emplace_back(timestamp, 0, 0, true);
    store.emplace_back(timestamp + 1000, 1, 1, false);
    store.emplace_back(timestamp + 2000, 2, 2, false);
    store.emplace_back(timestamp + 3000, 3, 3, true);

    // Perform time-based slicing of event store, the output event store "sliced" will contain
    // the second and third events from above. The end timestamp (second argument) is 2001, since start
    // timestamp (first argument) is inclusive and timestamp is exclusive, so 1 is added.
    const dv::EventStore sliced = store.sliceTime(timestamp + 1000, timestamp + 2001);

    // This should print two events
    for (const dv::Event &ev : sliced) {
        std::cout << fmt::format("Sliced event [{}, {}, {}, {}]", ev.timestamp(), ev.x(), ev.y(), ev.polarity())
                  << std::endl;
    }

    return 0;
}
```

```python
import dv_processing as dv

# Initialize an empty store
store = dv.EventStore()
```

(continues on next page)

(continued from previous page)

```

# Get the current timestamp
timestamp = dv.now()

# Add some events into the event store
# This allocates and inserts events at the back, the function arguments are:
# timestamp, x, y, polarity
store.push_back(timestamp, 0, 0, True)
store.push_back(timestamp + 1000, 1, 1, False)
store.push_back(timestamp + 2000, 2, 2, False)
store.push_back(timestamp + 3000, 3, 3, True)

# Perform time-based slicing of event store, the output event store "sliced" will␣
↪contain
# the second and third events from above. The end timestamp (second argument) is 2001,
↪ since start
# timestamp (first argument) is inclusive and timestamp is exclusive, so 1 is added.
sliced = store.sliceTime(timestamp + 1000, timestamp + 2001)

# This should print two events
for ev in store:
    print(f"Sliced event [{ev.timestamp()}, {ev.x()}, {ev.y()}, {ev.polarity()}]")
```

3.1.2 Slicing Event Stores

dv::EventStore can be sliced by time or by the number of events. Slicing is a shallow operation, it does not copy any data. Slicing returns a new *dv::EventStore* that only references the data requested. The original store is unaffected.

Slicing by time

dv::EventStore implements event data slicing by time in an efficient way. By reusing the underlying packet structure, the slicing is performed with `O(log n)` time complexity. The following sample shows the usage of time based slicing functions:

C++

Python

```
#include <dv-processing/core/core.hpp>
#include <dv-processing/data/generate.hpp>

int main() {
    using namespace std::chrono_literals;

    // Generate 10 events with time range [10000; 20000]
    const auto store = dv::data::generate::uniformEventsWithinTimeRange(10000, 10ms,␣
↪cv::Size(100, 100), 10);

    // Get all events with timestamp above 12500, it will be 13000 and up
    dv::EventStore eventsAfterTimestamp = store.sliceTime(12'500);

    // Print the timestamp ranges
    std::cout << "1. " << eventsAfterTimestamp << std::endl;

```

(continues on next page)

(continued from previous page)

```cpp
    // Slice event within time range [12000; 16000); the end time is exclusive
    const dv::EventStore eventsInRange = store.sliceTime(12'000, 16'000);

    // Print the timestamp ranges; It will print that range is [12000; 15000] since end time is exclusive and
    // event at timestamp 16000 is not going to be included.
    std::cout << "2. " << eventsInRange << std::endl;

    return 0;
}
```

```python
import dv_processing as dv
from datetime import timedelta

# Generate 10 events with time range [10000; 20000]
store = dv.data.generate.uniformEventsWithinTimeRange(10000, timedelta(milliseconds=10), (100, 100), 10)

# Get all events with timestamp above 12500, it will be 13000 and up
eventsAfterTimestamp = store.sliceTime(12500)

# Print the timestamp ranges
print(f"1. {eventsAfterTimestamp}")

# Slice event within time range [12000; 16000); the end time is exclusive
eventsInRange = store.sliceTime(12000, 16000)

# Print the timestamp ranges; It will print that range is [12000; 15000] since end time is exclusive and
# event at timestamp 16000 is not going to be included.
print(f"2. {eventsInRange}")
```

Slicing by number of events

dv::EventStore provides slicing capabilities by an index and number of events. The first argument to the `dv::EventStore::slice()` method is a starting index from which the output slice starts, the second optional argument is the number of events to be sliced. If the number argument is not provided, the slice will contain all events from given index.

Following sample shows how to slice an event store within given indices of the underlying events:

C++

Python

```cpp
#include <dv-processing/core/core.hpp>
#include <dv-processing/data/generate.hpp>

int main() {
    using namespace std::chrono_literals;

    // Add 10 event with timestamps in range [10000; 20000]
    const auto store = dv::data::generate::uniformEventsWithinTimeRange(10000, 10ms, cv::Size(100, 100), 10);

```

(continues on next page)

(continued from previous page)

```cpp
    // Get all events beyond and including index 5
    dv::EventStore eventsAfterIndex = store.slice(5);
    std::cout << "1. " << eventsAfterIndex << std::endl;

    // Get 3 events starting with index 2
    dv::EventStore eventsInRange = store.slice(2, 3);
    std::cout << "2. " << eventsInRange << std::endl;

    // Use sliceBack to retrieve event from the end; this call will retrieve last 3 events
    dv::EventStore lastEvents = store.sliceBack(3);
    std::cout << "3. " << lastEvents << std::endl;

    return 0;
}
```

```python
import dv_processing as dv
from datetime import timedelta

# Generate 10 events with time range [10000; 20000]
store = dv.data.generate.uniformEventsWithinTimeRange(10000, timedelta(milliseconds=10), (100, 100), 10)

# Get all events beyond and including index 5
events_after_index = store.slice(5)
print(f"1. {events_after_index}")

# Get 3 events starting with index 2
events_in_range = store.slice(2, 3)
print(f"2. {events_in_range}")

# Use sliceBack to retrieve event from the end; this call will retrieve last 3 events
last_events = store.sliceBack(3)
print(f"3. {last_events}")
```

Combining multiple EventStores

Multiple instances of *dv::EventStore* can be added together to have a single object to access and manage it. Since *dv::EventStore* uses pointers to underlying event packet data, combining does not involve any deep data copies, the implementation takes over shared memory ownership instead. Following sample show how to efficiently combine multiple event stores into one:

C++

Python

```cpp
#include <dv-processing/core/core.hpp>
#include <dv-processing/data/generate.hpp>

int main() {
    using namespace std::chrono_literals;

    // Generate 10 events with timestamps in range [10000; 20000]
    const auto store1 = dv::data::generate::uniformEventsWithinTimeRange(10000, 10ms, cv::Size(100, 100), 10);
```

(continues on next page)

(continued from previous page)

```cpp

    // Generate second event store with 10 events with timestamps in range [20000; 29000] to the second store
    const auto store2 = dv::data::generate::uniformEventsWithinTimeRange(20000, 10ms, cv::Size(100, 100), 10);

    // Final event store which will contain all events
    dv::EventStore finalStore;

    // Add the events into the final store; this operation is shallow, so no data copies
    // are performed, but the underlying data has shared ownership between all stores
    finalStore.add(store1);
    finalStore.add(store2);

    // Print specific information on what we contain in the event store
    std::cout << finalStore << std::endl;

    return 0;
}
```

```python
import dv_processing as dv
from datetime import timedelta

# Generate 10 events with timestamps in range [10000; 20000]
store1 = dv.data.generate.uniformEventsWithinTimeRange(10000, timedelta(milliseconds=10), (100, 100), 10)

# Generate second event store with 10 events with timestamps in range [20000; 29000] to the second store
store2 = dv.data.generate.uniformEventsWithinTimeRange(20000, timedelta(milliseconds=10), (100, 100), 10)

# Final event store which will contain all events
final_store = dv.EventStore()

# Add the events into the final store; this operation is shallow, so no data copies
# are performed, but the underlying data has shared ownership between all stores
final_store.add(store1)
final_store.add(store2)

# Print specific information on what we contain in the final event store
print(f"{final_store}")
```

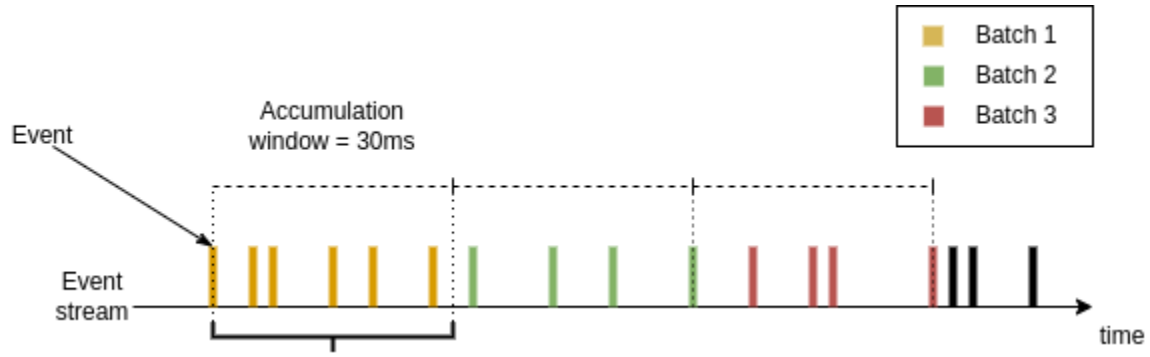
3.2 Event stream slicing

Events are a sparse representation for brightness changes registered by an event camera. It is useful to group incoming events from a stream in certain time-windows or by a certain number. The dv-processing library provides an efficient, yet powerful tools to approach continuous slicing of events. Following chapters will provide samples on how to use available event stream slicer implementations to group events in specific batches.

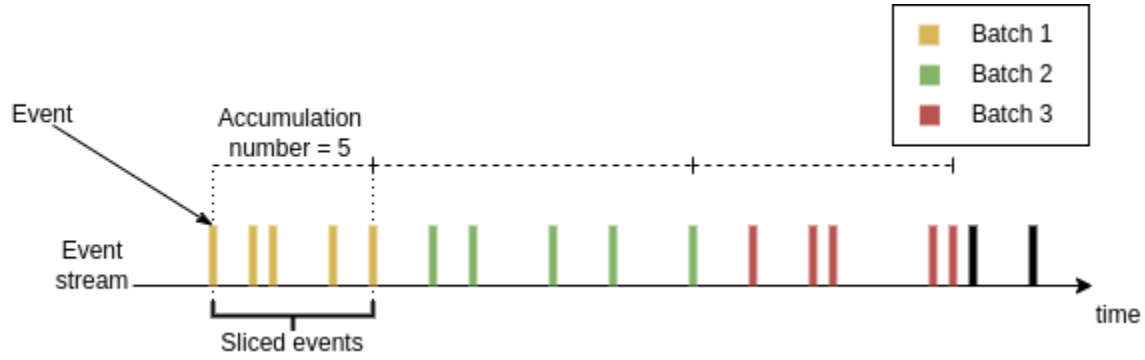
3.2.1 EventStreamSlicer

dv::EventStreamSlicer implements event slicing for a single event stream. Incoming events are passed into the slicer using the `accept()` method, the underlying implementation applies required slicing approach and resulting sliced events are passed into registered callback methods. The slicing can be performed using fixed size time-windows or fixed size number of events. According callback function can be registered using `dv::EventStreamSlicer::doEveryNumberOfEvents()` or `dv::EventStreamSlicer::doEveryTimeInterval()` methods. The registered callbacks and their calling parameters can also be modified later on.

The approach can be visualized by looking at a stream of events coming from a camera as a time series. The slicing can be performed by splitting the stream by a fixed time window as shown in a diagram below:



The other approach is to slice by a fixed number of events instead, as shown in the diagram below:



Slicing a stream

Following sample shows how to perform time-based slicing of an event stream:

C++

Python

```cpp
#include <dv-processing/core/core.hpp>
#include <dv-processing/data/generate.hpp>

int main() {
    // Use this namespace to enable literal time expression from the chrono library
    using namespace std::chrono_literals;

    // Initialize slicer, it will have no jobs at this time
    dv::EventStreamSlicer slicer;

    // Register this method to be called every 33 millisecond worth of event data
    slicer.doEveryTimeInterval(33ms, [](const dv::EventStore &events) {
        std::cout << "* Received events time-based slicing: " << events << std::endl;
    });

    // Register this method to be called every 100 events
    slicer.doEveryNumberOfElements(100, [](const dv::EventStore &events) {
        std::cout << "# Received events in number-based slicing: " << events <<␣
↪std::endl;
    });

    // Generate 1000 events within 2 second interval. These will be sliced correctly␣
↪by the slicer.
    const dv::EventStore store = dv::data::generate::uniformEventsWithinTimeRange(0,␣
↪2s, cv::Size(100, 100), 1000);

    // Now push the store into the slicer, the data contents within the store
    // can be arbitrary, the slicer implementation takes care of correct slicing
    // algorithm and calls the previously registered callbacks accordingly.
    slicer.accept(store);

    return 0;
}
```

```python
import dv_processing as dv
from datetime import timedelta

# Initialize slicer, it will have no jobs at this time
slicer = dv.EventStreamSlicer()


def print_time_interval(events: dv.EventStore):
    # Print the time duration received by this method
    print(f"* Received events time-based slicing: {events}")


# Register this method to be called every 33 millisecond worth of event data
slicer.doEveryTimeInterval(timedelta(milliseconds=33), print_time_interval)


```

(continues on next page)

(continued from previous page)

```python
def print_event_number(events: dv.EventStore):
    # Print the number of events received here
    print(f"# Received events in number-based slicing: {events}")


# Register this method to be called every 100 events
slicer.doEveryNumberOfEvents(100, print_event_number)

# Generate 1000 events within 2 second interval. These will be sliced correctly by the slicer.
store = dv.data.generate.uniformEventsWithinTimeRange(0, timedelta(seconds=2), (100, 100), 1000)

# Now push the store into the slicer, the data contents within the store
# can be arbitrary, the slicer implementation takes care of correct slicing
# algorithm and calls the previously registered callbacks accordingly.
slicer.accept(store)
```

Modifying slicing parameters

Slicing parameters can also be modified during runtime if needed. Following code sample is a modified version of the sample from previous chapter, which shows how to use the parameter modification methods:

C++

Python

```cpp
#include <dv-processing/core/core.hpp>
#include <dv-processing/data/generate.hpp>

int main() {
    // Use this namespace to enable literal time expression from the chrono library
    using namespace std::chrono_literals;

    // Initialize slicer, it will have no jobs at this time
    dv::EventStreamSlicer slicer;

    // Register this method to be called every 33 millisecond worth of event data
    int timeJobId = slicer.doEveryTimeInterval(33ms, [](const dv::EventStore &events) {
        std::cout << "* Received events time-based slicing: " << events << std::endl;
    });

    // Register this method to be called every 100 events
    int numberJobId = slicer.doEveryNumberOfElements(100, [](const dv::EventStore &events) {
        std::cout << "# Received events in number-based slicing: " << events << std::endl;
    });

    // Implement data generation; The following loop will generate 10 packets of events,
    // each containing 100 events within 20 millisecond duration.
    for (int i = 0; i < 10; i++) {
        // Generate 100 events within 20 millisecond interval. These will be sliced correctly by the slicer.
```

(continues on next page)

(continued from previous page)

```
        const auto store = dv::data::generate::uniformEventsWithinTimeRange(i * 20'000, 20ms, cv::Size(100, 100), 100);

        // Now push the store into the slicer, the data contents within the store
        // can be arbitrary, the slicer implementation takes care of correct slicing
        // algorithm and calls the previously registered callbacks accordingly.
        slicer.accept(store);

        // When a packet with index 5 is reached, modify the parameters
        if (i == 5) {
            // Modify time range to 10 milliseconds instead of 33
            slicer.modifyTimeInterval(timeJobId, 10ms);
            // Modify number to 200 instead of 100
            slicer.modifyNumberInterval(numberJobId, 200);
        }
    }
}
```

```
import dv_processing as dv
from datetime import timedelta

# Initialize slicer, it will have no jobs at this time
slicer = dv.EventStreamSlicer()


def print_time_interval(events: dv.EventStore):
    # Print the time duration received by this method
    print(f"* Received event with {events.duration()} duration in time-based slicing")


# Register this method to be called every 33 millisecond worth of event data
time_job_id = slicer.doEveryTimeInterval(timedelta(milliseconds=33), print_time_interval)


def print_event_number(events: dv.EventStore):
    # Print the number of events received here
    print(f"# Received {events.size()} events in number-based slicing")


# Register this method to be called every 100 events
number_job_id = slicer.doEveryNumberOfEvents(100, print_event_number)

# Implement data generation; The following loop will generate 10 stores
# of events, each containing 100 events within 10 millisecond duration.
for i in range(10):
    # Generate 100 events within 20 millisecond interval. These will be sliced correctly by the slicer.
    store = dv.data.generate.uniformEventsWithinTimeRange(i * 20000, timedelta(milliseconds=20), (100, 100), 100)

    # Now push the store into the slicer, the data contents within the store
    # can be arbitrary, the slicer implementation takes care of correct slicing
    # algorithm and calls the previously registered callbacks accordingly.
    slicer.accept(store)

```

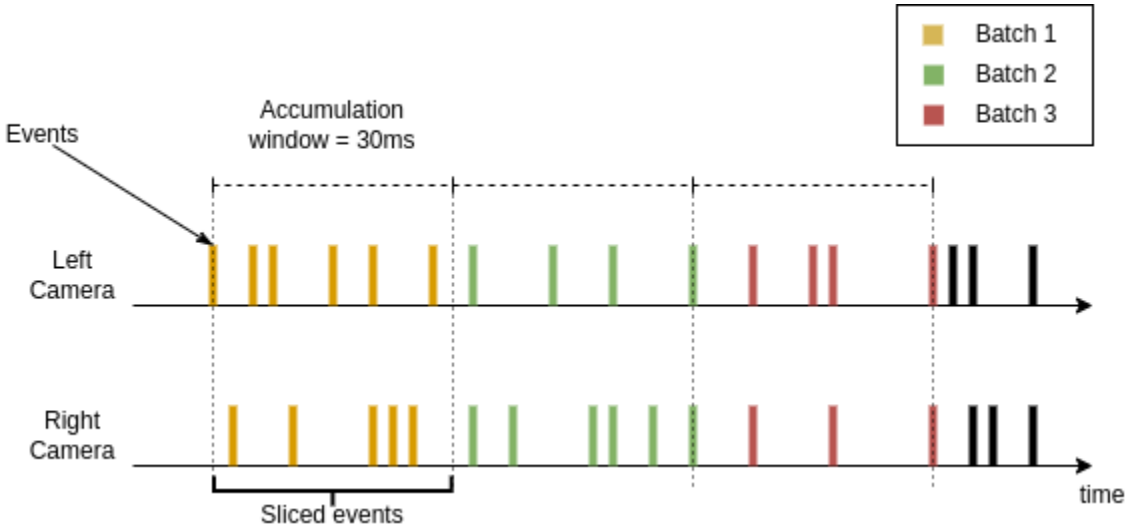
(continues on next page)

(continued from previous page)

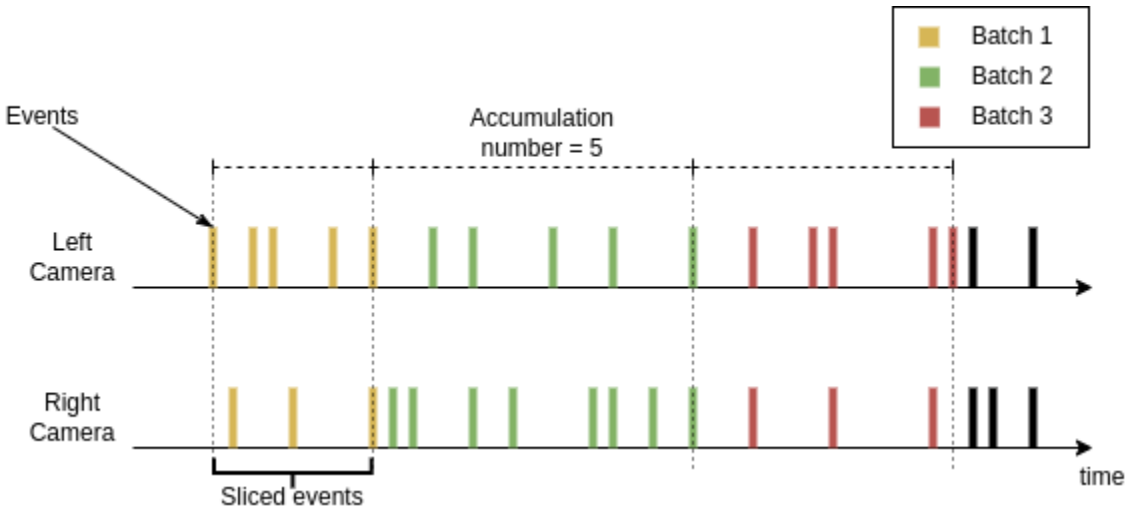
```
    # When a packet with index 5 is reached, modify the parameters
    if i == 5:
        # Modify time range to 10 milliseconds instead of 33
        slicer.modifyTimeInterval(time_job_id, timedelta(milliseconds=10))
        # Modify number to 200 instead of 100
        slicer.modifyNumberInterval(number_job_id, 200)
```

3.2.2 Stereo stream slicer

A dual stream slicer is required to perform synchronized time slicing from a stereo camera setup. This is implemented in the *dv::StereoEventStreamSlicer* class. It implements stereo event stream slicing by applying regular stream slicing on one of the input streams and performing time-based slicing within the same time-window. An illustration of the approach for slicing by time is displayed below:



Since the input data can be coming at different rates, in case of slicing by number, the according event from right camera are selected within the same time window. This case is shown below:



In the image above, the slicer is assumed to have a setting to slice every 5 events, these events are sliced from the left camera and their exact timestamps are used to slice events from the right camera.

A sample code on how to use stereo event stream slicer with a live camera is shown below:

C++

Python

```cpp
#include <dv-processing/core/stereo_event_stream_slicer.hpp>
#include <dv-processing/io/stereo_capture.hpp>

int main() {
    using namespace std::chrono_literals;

    // Discover connected camera to the system
    const auto cameras = dv::io::discoverDevices();
    if (cameras.size() < 2) {
        throw dv::exceptions::RuntimeError("Unable to discover two cameras");
    }

    // Open the cameras, just use first detected cameras
    dv::io::StereoCapture stereo(cameras[0], cameras[1]);

    // Initialize a stereo stream slicer
    dv::StereoEventStreamSlicer slicer;

    // Register a job to be performed every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [](const dv::EventStore &leftEvents, const dv::EventStore &rightEvents) {
        // Print durations for time-based slicing callback
        std::cout << fmt::format(
            "* Received events with duration: left[{}] - right[{}]", leftEvents.duration(), rightEvents.duration())
                  << std::endl;
    });

    // Register a job to be performed every 1000 events
    slicer.doEveryNumberOfEvents(
        // Here we receive events from two camera, time-synchronized
        1000, [](const dv::EventStore &leftEvents, const dv::EventStore &rightEvents) {
            // Print event store sizes for number based slicing callback
            std::cout << fmt::format("# Received events in number-based slicing, counts: left[{}] - right[{}]",
                leftEvents.size(), rightEvents.size())
                      << std::endl;
        });

    // Continue the loop while both cameras are connected
    while (stereo.left.isRunning() && stereo.right.isRunning()) {
        // Initialize
        dv::EventStore left, right;

        // Handle left camera events
        if (const auto batch = stereo.left.getNextEventBatch(); batch.has_value()) {
            left = *batch;
        }

        // Handle right camera events
        if (const auto batch = stereo.right.getNextEventBatch(); batch.has_value()) {
            right = *batch;
```

(continues on next page)

(continued from previous page)

```
        }

        // Pass all events into the slicer
        slicer.accept(left, right);
    }

    return 0;
}
```

```
import dv_processing as dv
from datetime import timedelta

# Discover connected camera to the system
cameras = dv.io.discoverDevices()
if len(cameras) < 2:
    raise RuntimeError("Unable to discover two cameras")

# Open the cameras, just use first detected cameras
stereo = dv.io.StereoCapture(cameras[0], cameras[1])

# Initialize a stereo stream slicer
slicer = dv.io.StereoEventStreamSlicer()


# Callback method for time based slicing
def print_time_interval(left_events: dv.EventStore, right_events: dv.EventStore):
    # Print the time duration received by this method
    print(f"* Received events with duration: left[{left_events.duration()}] - right[
↪{right_events.duration()}]")


# Register a job to be performed every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), print_time_interval)


# Callback method for number based slicing
def print_event_number(left_events: dv.EventStore, right_events: dv.EventStore):
    # Print the number of events received here
    print(
        f"# Received events in number-based slicing, counts: left[{left_events.size()}
↪] - right[{right_events.size()}]")


# Register this method to be called every 1000 events
slicer.doEveryNumberOfEvents(1000, print_event_number)

# Continue the loop while both cameras are connected
while stereo.left.isRunning() and stereo.right.isRunning():
    # Initialize empty stores
    left = dv.EventStore()
    right = dv.EventStore()

    # Receive packet from left camera
    leftPacket = stereo.left.getNextEventBatch()
    # Assign the packet if some data was received
    if leftPacket is not None:
```

(continues on next page)

(continued from previous page)

```
        left = leftPacket

    # Receive packet from right camera
    rightPacket = stereo.right.getNextEventBatch()
    # Assign the packet if some data was received
    if rightPacket is not None:
        right = rightPacket

    # Pass all events into the slicer
    slicer.accept(left, right)
```

3.2.3 Generic data stream slicing

Event stream slicers described previously are intended for efficient and specific slicing approach to event stream data. The dv-processing library also provides a generic time-series data stream slicer class `dv::MultiStreamSlicer` that applies the same approach, but supports arbitrary number of streams and arbitrary data-types. The only requirement for data types is the timestamp data, that can be accessed through a predefined API. The predefined API requirement is implemented using C++20 concepts and templates, so the requirement can be satisfied by any type that can provide a microsecond timestamp expressed in a signed 64-bit integer.

The `dv::MultiStreamSlicer` class provides same slicing capabilities with methods allowing to slice by time `dv::MultiStreamSlicer::doEveryTimeInterval()` or by number `dv::MultiStreamSlicer::doEveryNumberOfElements()`. Since the `dv::MultiStreamSlicer` supports an arbitrary number of streams, the streams are managed by assigning a unique string name to each stream by using the `dv::MultiStreamSlicer::addStream()` method. By default, the slicer accepts the timestamped data types from the DV flatbuffer type system, such as: *`dv::EventPacket`*, *`dv::IMUPacket`*, and other packet data types. If a data type is provided without a container, it can be used with STL containers, such as `std::vector` or dv-processing provided *`dv::cvector`*. In a case an image frame *`dv::Frame`* stream needs to be sliced, it has to be wrapped in a container, e.g. using *`dv::cvector`*: `dv::cvector<dv::Frame>`. Such a contained and timestamped data can be sliced alongside other data type streams.

To be clear about how the `MultiStreamSlicer` manages data slicing, it uses a convention of "main stream" and "secondary streams". Main stream is declared in the constructor of the class and all other stream are added using the `MultiStreamSlicer::addStream()` method. The main stream is the driving data stream slicing; while secondary streams are following the time-ranges that are sliced from the main stream, similar to how *`dv::StereoEventStreamSlicer`* works.

Following sample shows the use of the `MultiStreamSlicer` to synchronously slice incoming frame and event streams from a DAVIS346 camera and show a preview. The preview below

```
#include <dv-processing/core/multi_stream_slicer.hpp>
#include <dv-processing/io/camera_capture.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

int main() {
    // Open a DAVIS camera, the sample will slice incoming frames and events␣
↪synchronously using the generic data
    // stream slicer. The camera name is an empty string to open any DAVIS camera␣
↪detected on the system
    dv::io::CameraCapture camera("", dv::io::CameraCapture::CameraType::DAVIS);

```

(continues on next page)

(continued from previous page)

```
    // Declare the main stream type to be dv::cvector<dv::Frame> with stream name "frames"
    dv::MultiStreamSlicer<dv::cvector<dv::Frame>> slicer("frames");

    // Add a secondary stream of events
    slicer.addStream<dv::EventStore>("events");

    // It's possible to add additional secondary streams for slicing here, e.g. adding a trigger stream
    // would look like:
    // slicer.addStream<dv::TriggerPacket>("triggers");

    // Use visualizer to overlay events on a frame
    dv::visualization::EventVisualizer visualizer(camera.getEventResolution().value(), dv::visualization::colors::white,
        dv::visualization::colors::green, dv::visualization::colors::red);

    // Declare display windows for frames
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Register a job to be performed every frame, the event will be sliced in-between the main stream frames
    slicer.doEveryNumberOfElements(
        // Here we receive time-synchronized frames and events; sliced data is provided in a std::unordered_map
        // for easy access using the stream name.
        1, [&visualizer](const auto &data) {
            // Extract events and pass them to the slicer, data is retrieved using the helper method `get()`
            // which accepts the stream name and the type. Stream name and type has to match, otherwise
            // an exception will be thrown
            const auto events = data.template get<dv::EventStore>("events");

            // Retrieve frames, although we get one frame per slice, it is stored in the configured container
            const auto frames = data.template get<dv::cvector<dv::Frame>>("frames");

            // The container is non-empty, we expect only one frame in the container, so just display
            // the first frame in the container
            const dv::Frame &frame = frames.at(0);

            // Convert the frame into a grayscale image for overlay preview
            cv::Mat preview;
            if (frame.image.channels() == 3) {
                // The image is already grayscale, no conversion is needed
                preview = frame.image;
            }
            else {
                // The image coming from the camera is multichannel image, so we can safely assume
                // it's a grayscale image, convert it into BGR for overlay drawing
                cv::cvtColor(frame.image, preview, cv::COLOR_GRAY2BGR);
            }

            // Overlay events on top of preview image
```

(continues on next page)

(continued from previous page)

```
            visualizer.generateImage(events, preview);

            // Display the overlayed image
            cv::imshow("Preview", preview);

            // If escape button is pressed (code 27 is escape key), exit the program cleanly
            if (cv::waitKey(2) == 27) {
                exit(0);
            }
        });

    // Continue the loop while both cameras are connected
    while (camera.isRunning()) {
        // Handle events
        if (const auto events = camera.getNextEventBatch(); events.has_value()) {
            slicer.accept("events", *events);
        }

        // Handle frames
        if (const auto frame = camera.getNextFrame(); frame.has_value()) {
            slicer.accept("frames", *frame);
        }
    }

    return 0;
}
```

The multi stream slicer is also available in python, but due to missing template functionality in python, the python version is more limited. `dv.EventMultiStreamSlicer` class is provided in python, which only supports events as main stream and has slightly different API, but it supports arbitrary number of secondary streams, which can be of any type that is built-in in dv-processing. Secondary streams can be added using named methods for types.

The example below shows the use of the multi-stream slicer in python which uses events as main stream and slices frames as secondary stream:

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open the camera, just use first detected DAVIS camera
camera = dv.io.CameraCapture("", dv.io.CameraCapture.CameraType.DAVIS)

# Initialize a multi-stream slicer
slicer = dv.EventMultiStreamSlicer("events")

# Add a frame stream to the slicer
slicer.addFrameStream("frames")

# Initialize a visualizer for the overlay
visualizer = dv.visualization.EventVisualizer(camera.getEventResolution(), dv.visualization.colors.white(),
                                              dv.visualization.colors.green(), dv.visualization.colors.red())

# Create a window for image display
cv.namedWindow("Preview", cv.WINDOW_NORMAL)
```

(continues on next page)

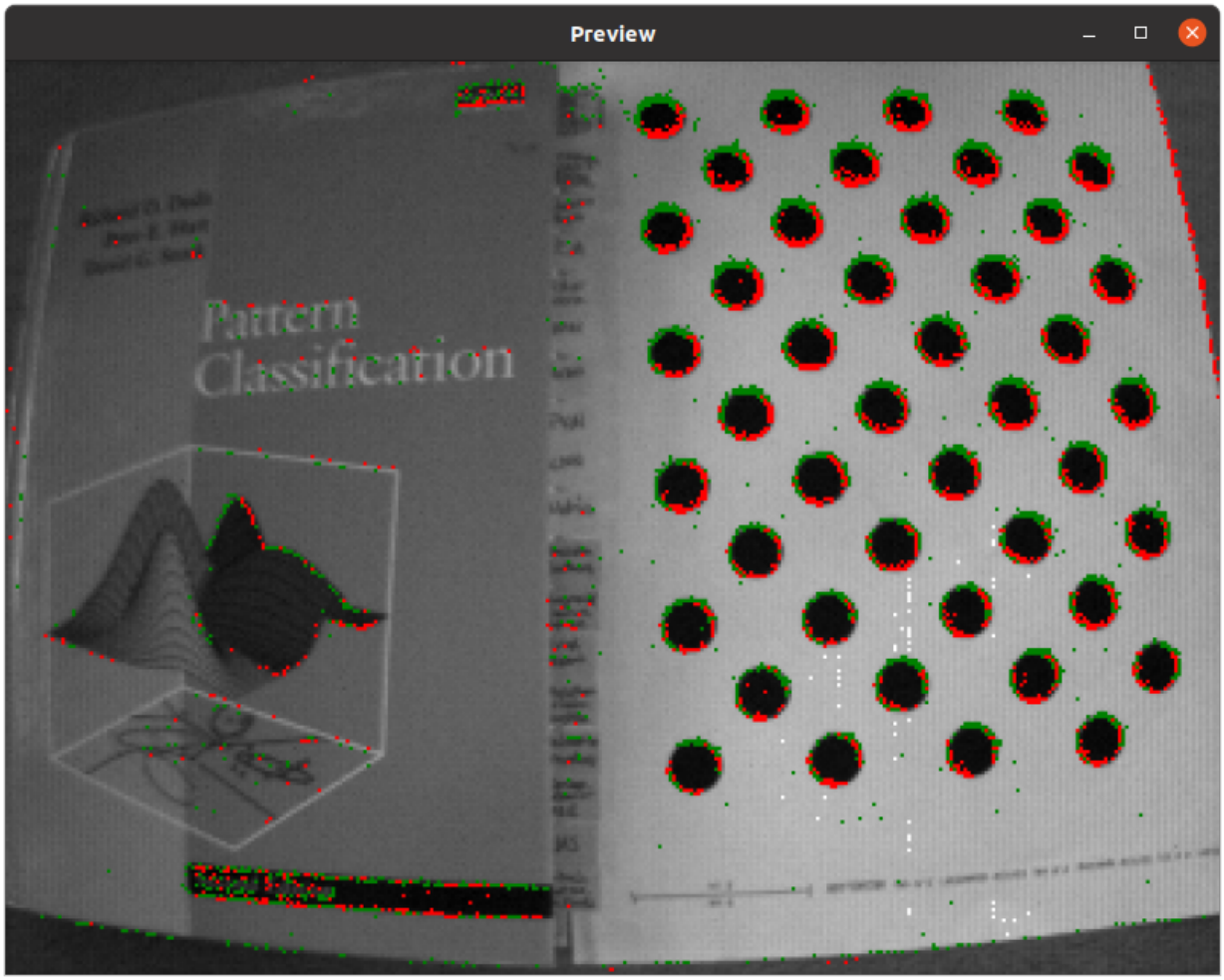


Fig. 2: Output from the multi stream sample usage - frames with synchronized events overlayed in the frame image.

(continued from previous page)

```python


# Callback method for time based slicing
def display_preview(data):
    # Retrieve frame data using the named method and stream name
    frames = data.getFrames("frames")

    # Retrieve event data
    events = data.getEvents("events")

    # Retrieve and color convert the latest frame of retrieved frames
    latest_image = None
    if len(frames) > 0:
        if len(frames[-1].image.shape) == 3:
            # We already have colored image, no conversion
            latest_image = frames[-1].image
        else:
            # Image is grayscale, convert to color (BGR image)
            latest_image = cv.cvtColor(frames[-1].image, cv.COLOR_GRAY2BGR)
    else:
        return

    # Generate a preview and show the final image
    cv.imshow("Preview", visualizer.generateImage(events, latest_image))

    # If escape button is pressed (code 27 is escape key), exit the program cleanly
    if cv.waitKey(2) == 27:
        exit(0)


# Register a job to be performed every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), display_preview)

# Continue the loop while both cameras are connected
while camera.isRunning():
    events = camera.getNextEventBatch()
    if events is not None:
        slicer.accept("events", events)

    frame = camera.getNextFrame()
    if frame is not None:
        slicer.accept("frames", [frame])
```

This sample will result in similar output to the C++ version, but it rather synchronizes events to frames, so output of the samples are comparable, but not exact.

3.3 Event accumulation

Events are a sparse representation of brightness changes measured by the pixels on a camera sensor. Accumulation is a method applied to events to generate a frame representation of events. The dv-processing library provides a few highly optimized algorithm implementations to perform event accumulation and achieve different frame-like representation of the events.

3.3.1 Definitions

Certain definitions are used in this chapter and within accumulator API. Following is a list of specific definitions and their meaning within context of the accumulator:

- Potential - normalized pixel brightness value in some floating point range; it is an internal representation range for brightness that can be scaled into other brightness representations;
- Contribution - a numeric value that an event contributes to the brightness of pixel;
- Decay - pixel brightness correction applied over time when no events contribute to its brightness;
- Neutral potential - a default pixel brightness without any contribution;
- Minimum / maximum potential - limits for potential value representation.

3.3.2 Accumulator

The `dv::Accumulator` is a generalized implementation of a few accumulation algorithms that can be configured using class methods. Following chapter will describe the configuration options available for the accumulator. Final chapter of this section will provide a code sample that shows the use of all these configuration options for accumulation of events from a live camera.

Decay function and decay param

One of *None*, *Linear*, *Exponential*, *Step*. Defines the data degradation function that should be applied to the image. For each function, the *Decay param* setting assumes a different function:

Function	Enum value	*Decay param* function	Explanation
None	`Decay::NONE`	No function	Does not apply any decay
Linear	`Decay::LINEAR`	The slope `a` of the linear function, in *intensity per microsecond*	Assume intensity of a pixel is `I0` at time `0`, this function applies a linear decay of the form of `I0 - (t * decayparam)` or `I0 + (t * decayparam)` until the value hits the value specified in *neutral potential*
Exponential	`Decay::EXPONENTIAL`	The time constant `tau` of the exponential function in *microseconds*	Assume intensity of a pixel is `I0` at time `0`, this function applies an exponential decay of the form `I0 * exp(-(t/decayparam))`. The decay approaches a value of `neutralPotential` over time.
Step	`Decay::STEP`	No function	Set all pixel values to *neutral potential* after a frame is extracted.

The decay function can be set using `dv::Accumulator::setDecayFunction()` method.

Event Contribution

The contribution an event has onto the image. If an event arrives at a position `x`, `y`, the pixel value in the frame at `x`, `y` gets increased / decreased by the value of *Event contribution*, based on the events polarity.

Except:

- The resulting pixel value would be higher than *Max potential*, the value gets set to *Max potential* instead
- The resulting pixel value would be lower than *Min potential*, the value gets set to *Min potential* instead
- The event polarity is negative, and *Ignore polarity* is enabled, then the event is counted positively

Event contribution can be set using the *dv::Accumulator::setEventContribution()* method.

Min potential / Max potential

Sets the minimum and maximum values a pixel can achieve. If the value of the pixel would reach higher or lower, it is capped at these values. These values are also used for normalization at the output. The frame the module generates is an `unsigned 8-bit` grayscale image, normalized between *Min potential* and *Max potential*. A pixel with the value *Min potential* corresponds to a pixel with the value `0` in the output frame. A pixel with the value *Max potential* corresponds to a pixel with the value `255` in the output frame.

Min potential and *Max potential* can be set using *dv::Accumulator::setMinPotential()* and *dv::Accumulator::setMaxPotential()* methods.

Neutral potential

This setting has different effects depending on the decay function:

Function	*Neutral potential* function
None	No function.
Linear	Pixel brightness value decays linearly into the *neutral potential* value.
Exponential	No function.
Step	Each pixel value is set to *neutral potential* after generating a frame.

Neutral potential can be set using *dv::Accumulator::setNeutralPotential()*.

Ignore polarity

If this value is set, all events act as if they had positive polarity. In this case, *Event contribution* is always taken positively. This can be used to generate edge images instead of an actual image reconstruction. This can be done by setting *Neutral potential* and *Min potential* to zeros.

This feature can be enabled using the *dv::Accumulator::setIgnorePolarity()* method.

Synchronous Decay

If this value is set, decay happens in continuous time for all pixels. In every frame, each pixel will be eagerly decayed to the time the image gets generated. If this value is not set, decay at the individual pixel only happens when the pixel receives an event. Decay is lazily evaluated at the pixel.

Note: Both decay regimes yield the same overall decay over time, just the time at which it is applied changes. This parameter does not have an effect for Step decay. Step decay is always synchronous at generation time.

Synchronous decay can be enabled using the *dv::Accumulator::setSynchronousDecay()* method.

Accumulating event from a camera

The following sample code show how to use *dv::Accumulator* together with *dv::EventStreamSlicer* and *dv::io::CameraCapture* to implement a pipeline that generates continuous stream of accumulated frames:

C++

Python

```cpp
#include <dv-processing/core/frame.hpp>
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    // Initialize an accumulator with some resolution
    dv::Accumulator accumulator(*capture.getEventResolution());

    // Apply configuration, these values can be modified to taste
    accumulator.setMinPotential(0.f);
    accumulator.setMaxPotential(1.f);
    accumulator.setNeutralPotential(0.5f);
    accumulator.setEventContribution(0.15f);
    accumulator.setDecayFunction(dv::Accumulator::Decay::EXPONENTIAL);
    accumulator.setDecayParam(1e+6);
    accumulator.setIgnorePolarity(false);
    accumulator.setSynchronousDecay(false);

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;
```

(continues on next page)

(continued from previous page)

```cpp

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&accumulator](const dv::EventStore &events) {
        // Pass events into the accumulator and generate a preview frame
        accumulator.accept(events);
        dv::Frame frame = accumulator.generateFrame();

        // Show the accumulated image
        cv::imshow("Preview", frame.image);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize an accumulator with some resolution
accumulator = dv.Accumulator(capture.getEventResolution())

# Apply configuration, these values can be modified to taste
accumulator.setMinPotential(0.0)
accumulator.setMaxPotential(1.0)
accumulator.setNeutralPotential(0.5)
accumulator.setEventContribution(0.15)
accumulator.setDecayFunction(dv.Accumulator.Decay.EXPONENTIAL)
accumulator.setDecayParam(1e+6)
accumulator.setIgnorePolarity(False)
accumulator.setSynchronousDecay(False)

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
```

(continues on next page)

(continued from previous page)

```python
def slicing_callback(events: dv.EventStore):
    # Pass events into the accumulator and generate a preview frame
    accumulator.accept(events)
    frame = accumulator.generateFrame()

    # Show the accumulated image
    cv.imshow("Preview", frame.image)
    cv.waitKey(2)


# Register a callback every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

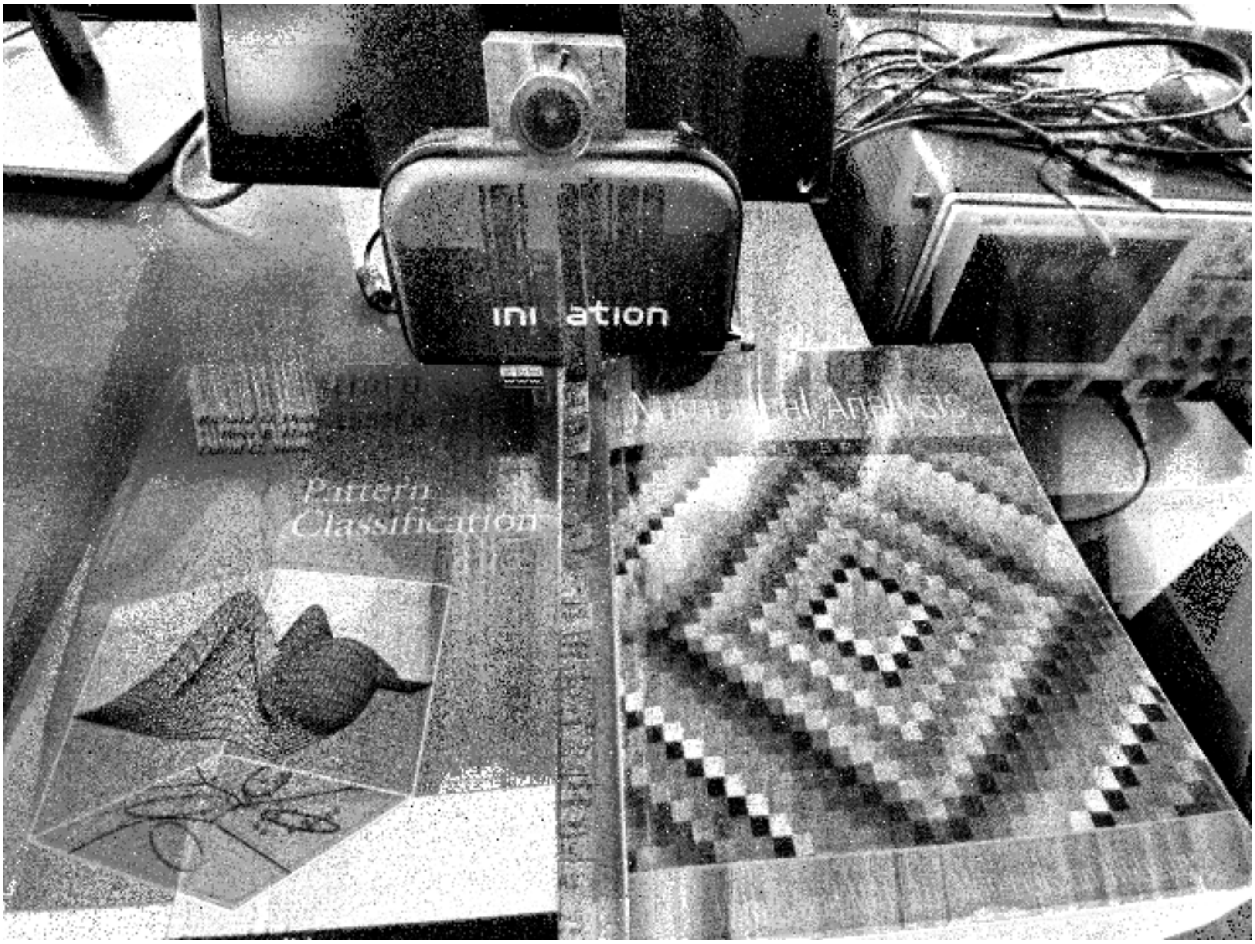


Fig. 3: Frame generated using `dv::Accumulator` class.

3.3.3 Edge accumulation

The dv-processing library provides a highly-optimized variant of accumulator for generating edge maps - *dv::EdgeMapAccumulator*. It was specifically optimizes for speed of execution, so it has only a minimal set of settings and supported features compared to *dv::Accumulator*.

Below is a table providing available parameters for the *dv::EdgeMapAccumulator*:

Parameter	Default value	Accepted values	Comment
Contribution	0.25	[0.0; 1.0]	Contribution potential for a single event.
Ignore polarity	true	boolea	All events are considered positive if enabled.
Neutral potential	0.0	[0.0; 1.0]	Neutral potential is the default pixel value when decay is disabled and the value that pixels decay into when decay is enabled.
Decay param	1.0	[0.0; 1.0]	This value defines how fast pixel values decay to neutral value. The bigger the value the faster the pixel value will reach neutral value. Decay is applied before each frame generation. The range for decay value is [0.0; 1.0], where 0.0 will not apply any decay and 1.0 will apply maximum decay value resetting a pixel to neutral potential at each generation (default behavior).

Following sample show the use of *dv::EdgeMapAccumulator* with *dv::EventStreamSlicer* and *dv::io::CameraCapture* to generate stream of edge images:

C++

Python

```cpp
#include <dv-processing/core/frame.hpp>
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

```

(continues on next page)

(continued from previous page)

```
    // Initialize an accumulator with some resolution
    dv::EdgeMapAccumulator accumulator(*capture.getEventResolution());

    // Apply configuration, these values can be modified to taste
    accumulator.setNeutralPotential(0.0f);
    accumulator.setEventContribution(0.25f);
    accumulator.setDecay(1.0);
    accumulator.setIgnorePolarity(true);

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&accumulator](const dv::EventStore &events) {
        // Pass events into the accumulator and generate a preview frame
        accumulator.accept(events);
        dv::Frame frame = accumulator.generateFrame();

        // Show the accumulated image
        cv::imshow("Preview", frame.image);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize an accumulator with some resolution
accumulator = dv.EdgeMapAccumulator(capture.getEventResolution())

# Apply configuration, these values can be modified to taste
accumulator.setNeutralPotential(0.0)
accumulator.setContribution(0.25)
accumulator.setNeutralPotential(1.0)
```

(continues on next page)

(continued from previous page)

```
accumulator.setIgnorePolarity(True)

# Initialize a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Pass events into the accumulator and generate a preview frame
    accumulator.accept(events)
    frame = accumulator.generateFrame()

    # Show the accumulated image
    cv.imshow("Preview", frame.image)
    cv.waitKey(2)


# Register callback to be performed every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

3.3.4 Event visualization

Accumulators, described in previous chapters are useful when image or edge representation of events is needed, and they are mostly useful to process events using typical image processing algorithms. *dv::visualization::EventVisualizer* class serves a purpose to perform simple event visualization. Instead of increasing or decreasing pixel brightness, the *dv::visualization::EventVisualizer* just performs color coding of pixel coordinates where an event was registered.

Following sample show the use of *dv::visualization::EventVisualizer* class to generate colored previews of events using *dv::EventStreamSlicer* and *dv::io::CameraCapture*:

C++

Python

```
#include <dv-processing/io/camera_capture.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

```

(continues on next page)

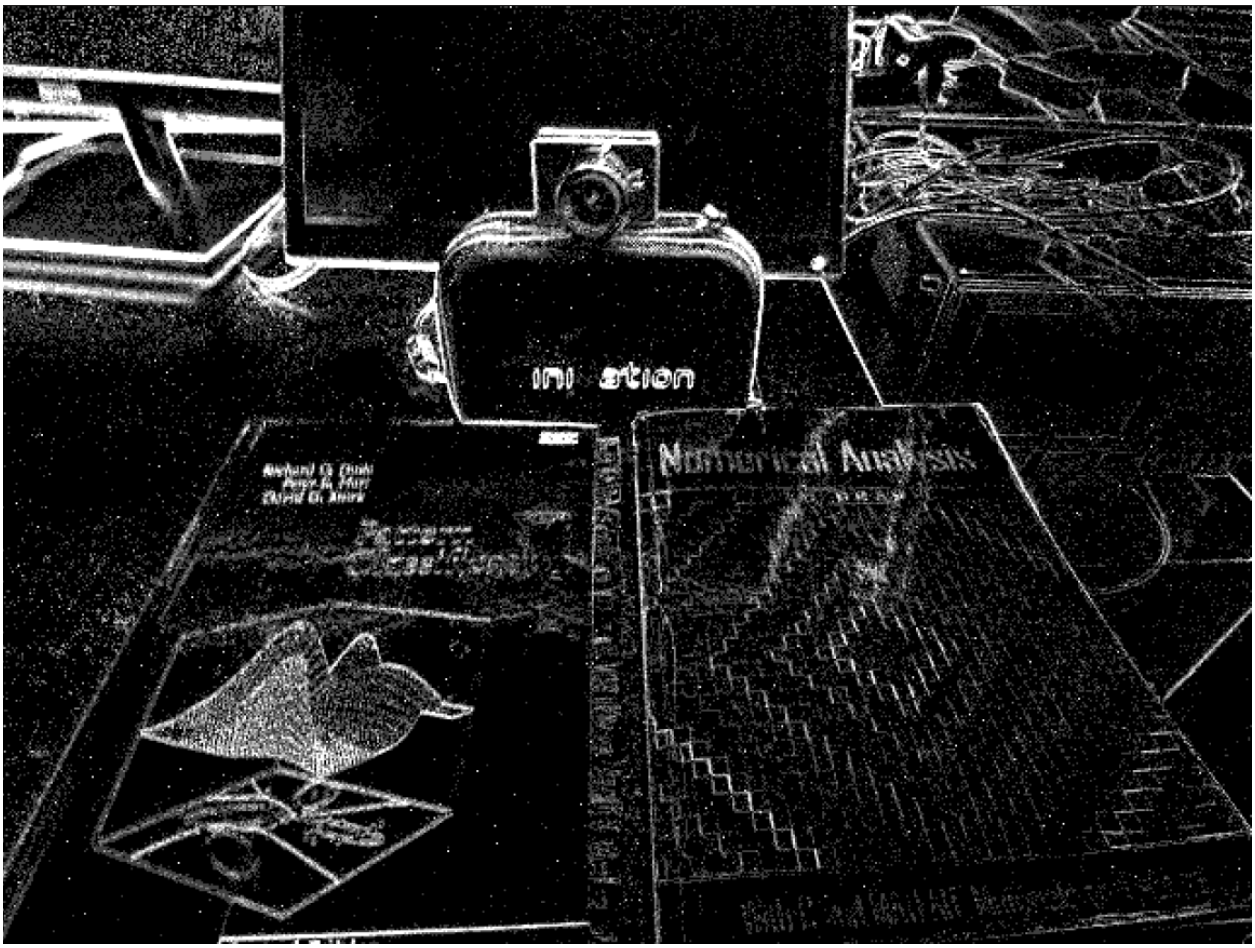


Fig. 4: Frame generated using `dv::EdgeMapAccumulator` class.

(continued from previous page)

```cpp
    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    // Initialize an accumulator with some resolution
    dv::visualization::EventVisualizer visualizer(*capture.getEventResolution());

    // Apply color scheme configuration, these values can be modified to taste
    visualizer.setBackgroundColor(dv::visualization::colors::white);
    visualizer.setPositiveColor(dv::visualization::colors::iniBlue);
    visualizer.setNegativeColor(dv::visualization::colors::darkGrey);

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&visualizer](const dv::EventStore &events) {
        // Generate a preview frame
        cv::Mat image = visualizer.generateImage(events);

        // Show the accumulated image
        cv::imshow("Preview", image);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

```

(continues on next page)

(continued from previous page)

```
# Initialize an accumulator with some resolution
visualizer = dv.visualization.EventVisualizer(capture.getEventResolution())

# Apply color scheme configuration, these values can be modified to taste
visualizer.setBackgroundColor(dv.visualization.colors.white())
visualizer.setPositiveColor(dv.visualization.colors.iniBlue())
visualizer.setNegativeColor(dv.visualization.colors.darkGrey())

# Initialize a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Generate a preview frame
    frame = visualizer.generateImage(events)

    # Show the accumulated image
    cv.imshow("Preview", frame)
    cv.waitKey(2)


# Register callback to be performed every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

3.3.5 Time surface

Time surface is an event representation in an image frame, except instead of representing event in pixel brightness, it represents event in a 2D image structure, but pixel value contains the latest event timestamp.

The timestamps representations can be normalized to retrieve an image representation of the time surface. It will represent the latest timestamps with the brightest pixel values.

Following sample show the use of *dv::TimeSurface* class to generate time surface previews of events using *dv::EventStreamSlicer* and *dv::io::CameraCapture*:

C++

Python

```
#include <dv-processing/core/frame.hpp>
#include <dv-processing/io/camera_capture.hpp>

```

(continues on next page)

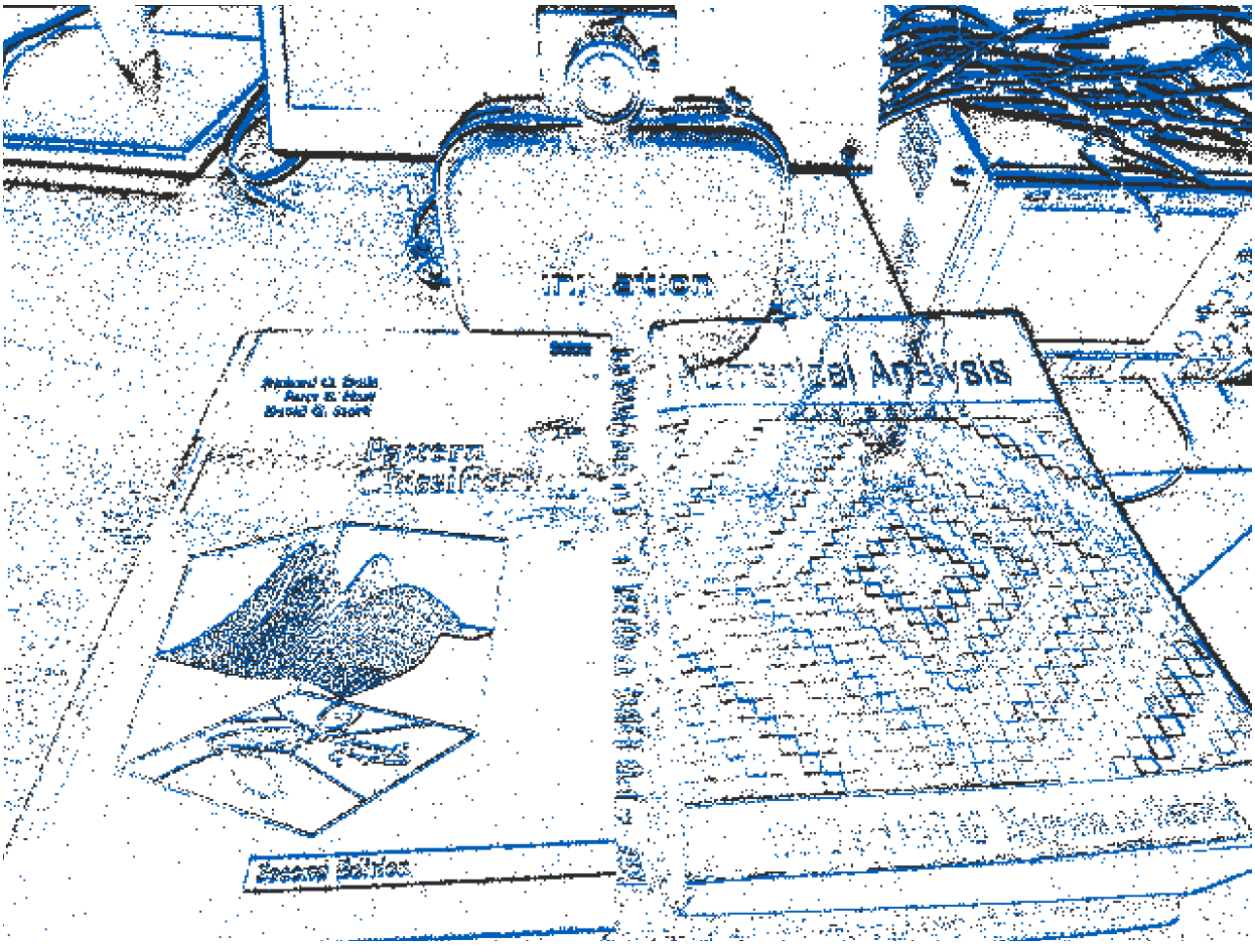


Fig. 5: Frame generated using `dv::visualization::EventVisualizer` class.

(continued from previous page)

```cpp
#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    // Initialize an accumulator with camera sensor resolution
    dv::TimeSurface surface(*capture.getEventResolution());

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&surface](const dv::EventStore &events) {
        // Pass the events to update the time surface
        surface.accept(events);

        // Generate a preview frame
        dv::Frame frame = surface.generateFrame();

        // Show the accumulated image
        cv::imshow("Preview", frame.image);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
```

(continues on next page)

(continued from previous page)

```python
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize an accumulator with camera sensor resolution
surface = dv.TimeSurface(capture.getEventResolution())

# Initialize a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Pass the events to update the time surface
    surface.accept(events)

    # Generate a preview frame
    frame = surface.generateFrame()

    # Show the accumulated image
    cv.imshow("Preview", frame.image)
    cv.waitKey(2)


# Register callback to be performed every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

3.3.6 Speed invariant time surface

Speed invariant time surface is a specific time surface variant that is more suitable for feature extraction, the implementation follows this paper: https://arxiv.org/pdf/1903.11332.pdf.

Following sample show the use of *dv::SpeedInvariantTimeSurface* class to generate time surface previews of events using *dv::EventStreamSlicer* and *dv::io::CameraCapture*:

C++

Python

```cpp
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
```

(continues on next page)

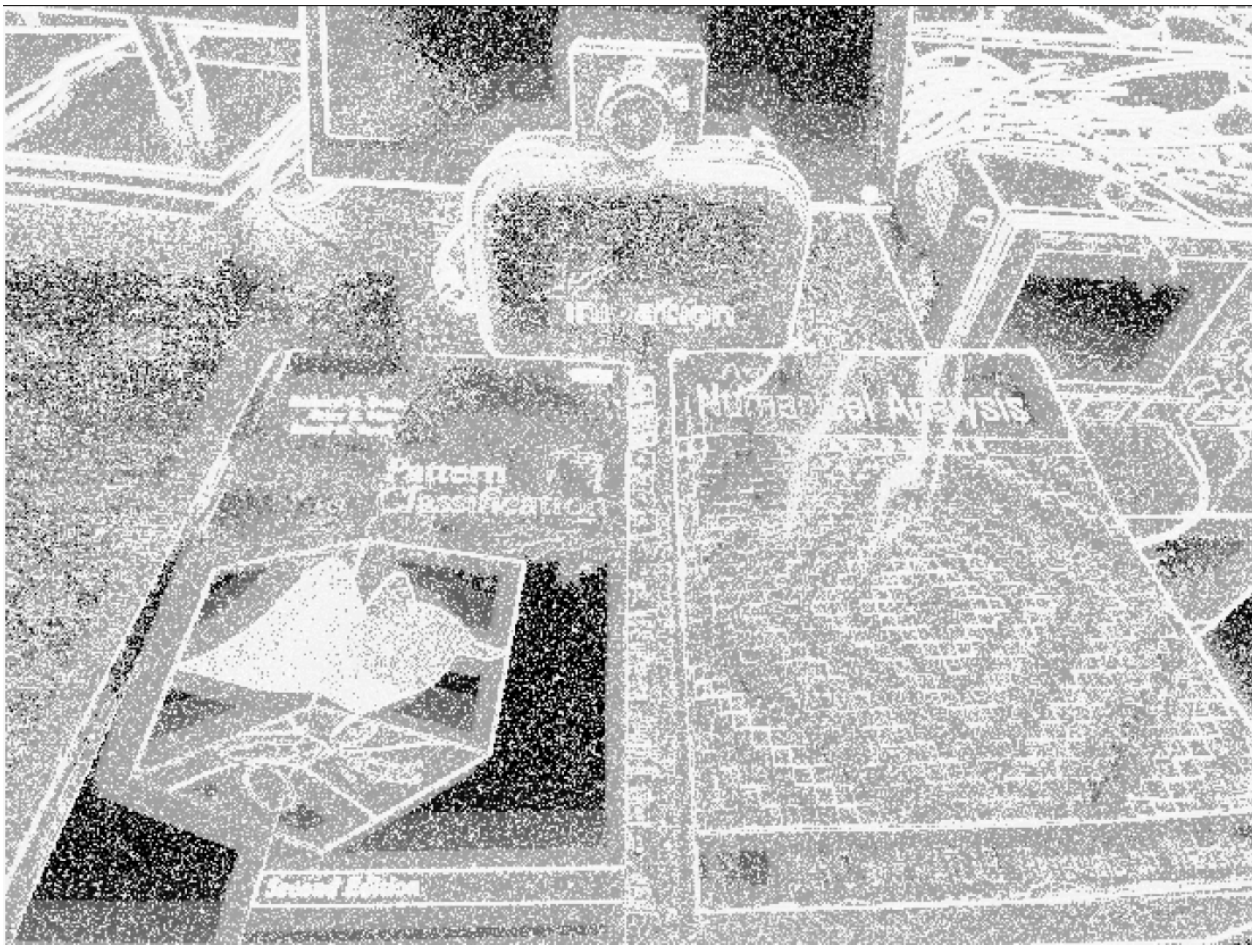


Fig. 6: Frame generated using `dv::TimeSurface` class.

(continued from previous page)

```cpp
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    // Initialize an accumulator with camera sensor resolution
    dv::SpeedInvariantTimeSurface surface(*capture.getEventResolution());

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&surface](const dv::EventStore &events) {
        // Pass the events to update the time surface
        surface.accept(events);

        // Generate a preview frame
        dv::Frame frame = surface.generateFrame();

        // Show the accumulated image
        cv::imshow("Preview", frame.image);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

```

(continues on next page)

(continued from previous page)

```
# Initialize an accumulator with camera sensor resolution
surface = dv.SpeedInvariantTimeSurface(capture.getEventResolution())

# Initialize a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Pass the events to update the time surface
    surface.accept(events)

    # Generate a preview frame
    frame = surface.generateFrame()

    # Show the accumulated image
    cv.imshow("Preview", frame.image)
    cv.waitKey(2)


# Register callback to be performed every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

3.3.7 Performance of available accumulators

The library performs benchmarking of available accumulation algorithms to ensure their best performance. Accumulators are also benchmarked on two metrics:

- Event throughput - measured in millions of (mega) events per second;
- Framerate - measured in generated frames per second.

The benchmarks are performed by generating a batch of events at uniformly random pixel coordinates on a VGA (640x480) resolution. Below are the results of running the benchmark on AMD Ryzen 7 3800X 8-Core Processor:

Accumulator type	Framerate (FPS)	Throughput (MegaEvent/s)
Accumulator	668	66.5
EdgeMapAccumulator	1767	149.1
EventVisualizer	785	78.3
TimeSurface	910	91.5
SpeedInvariantTimeSurface	370	36.8

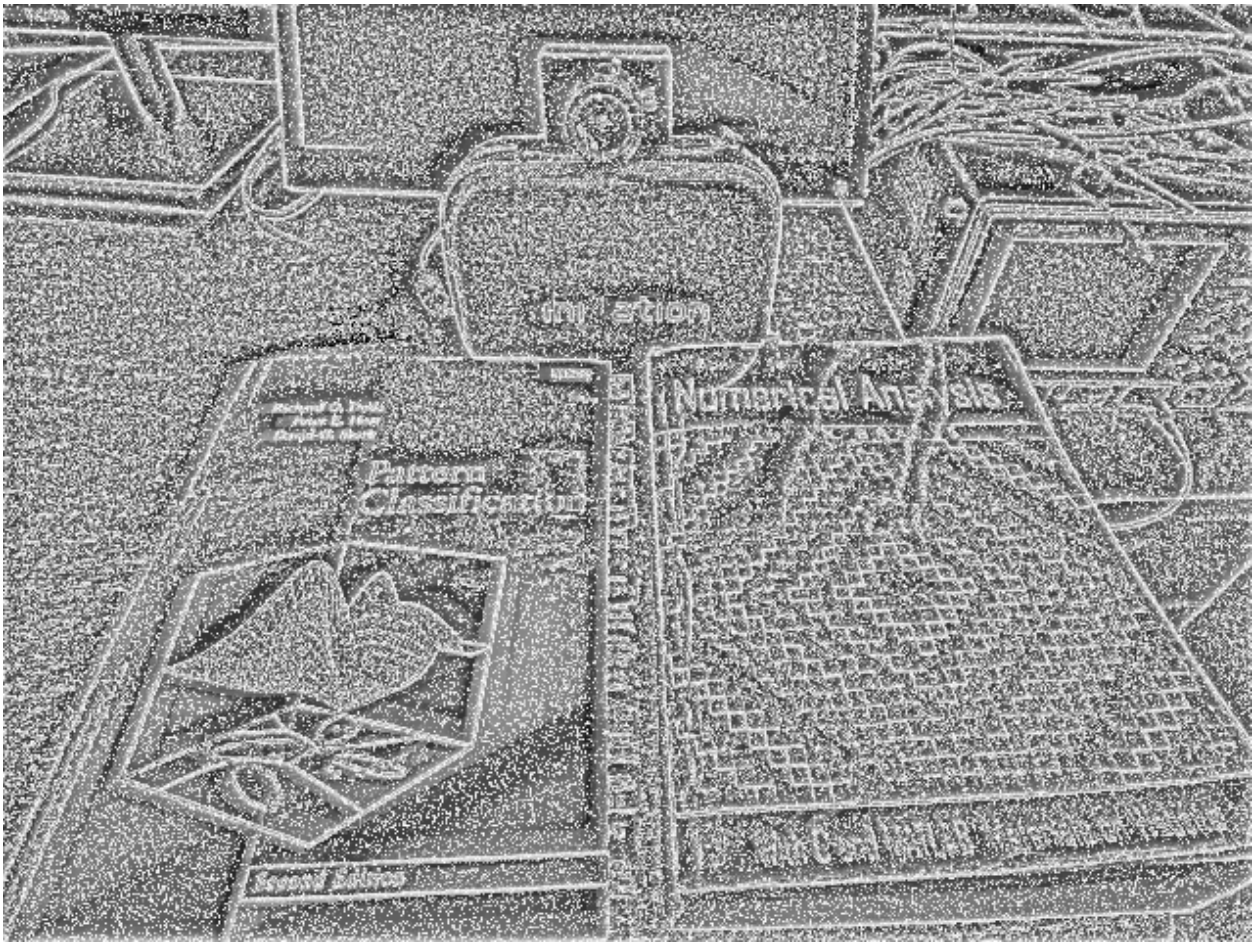


Fig. 7: Frame generated using `dv::SpeedInvariantTimeSurface` class.

3.4 Filtering events

Event stream coming from a camera can contain noise, which is caused by analog electrical circuitry used to compare brightness on each pixel. The dv-processing library provides algorithms that filter noise efficiently on an event stream. Additionally, filtering can be used to subsample events in region of interest, filter by polarity, apply masks to events. This tutorial covers available filter implementations and how to use them efficiently.

3.4.1 Implementation of filters

The library provides two main types of filters - noise and subsampling. Subsampling filters include polarity filters, region of interest, mask filters. The library provides two algorithms for filtering noise that can be found under namespace `dv::noise`. Below is a class hierarchy diagram for available noise filters:

All filter have a common programming pattern:

- Events are added to the filter instance using overloaded *dv::EventFilterBase::accept()* method.
- Input events are filtered and returned by calling overloaded *dv::EventFilterBase::generateEvents()* method.
- Internally events are filtered by calling a filter `retain()` method on each event and discarding events if a `false` is returned.

3.4.2 Noise filtering

This chapter describes the available event noise filters in the library.

Background activity noise filter

dv::noise::BackgroundActivityNoiseFilter - events are filtered based on short-term local neighborhood activity. If an event is "supported" by another event registered at local pixel neighborhood, that event is not considered noise and is added to the output events.

Following sample code shows the usage of the *dv::noise::BackgroundActivityNoiseFilter* to filter noise:

C++

Python

```cpp
#include <dv-processing/data/generate.hpp>
#include <dv-processing/noise/background_activity_noise_filter.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Hardcoded VGA resolution
    const cv::Size resolution(640, 480);

    // Initializing input events with uniformly distributed events which represent␣
↪noise
    dv::EventStore events = dv::data::generate::uniformEventsWithinTimeRange(0, 10ms,␣
```

(continues on next page)

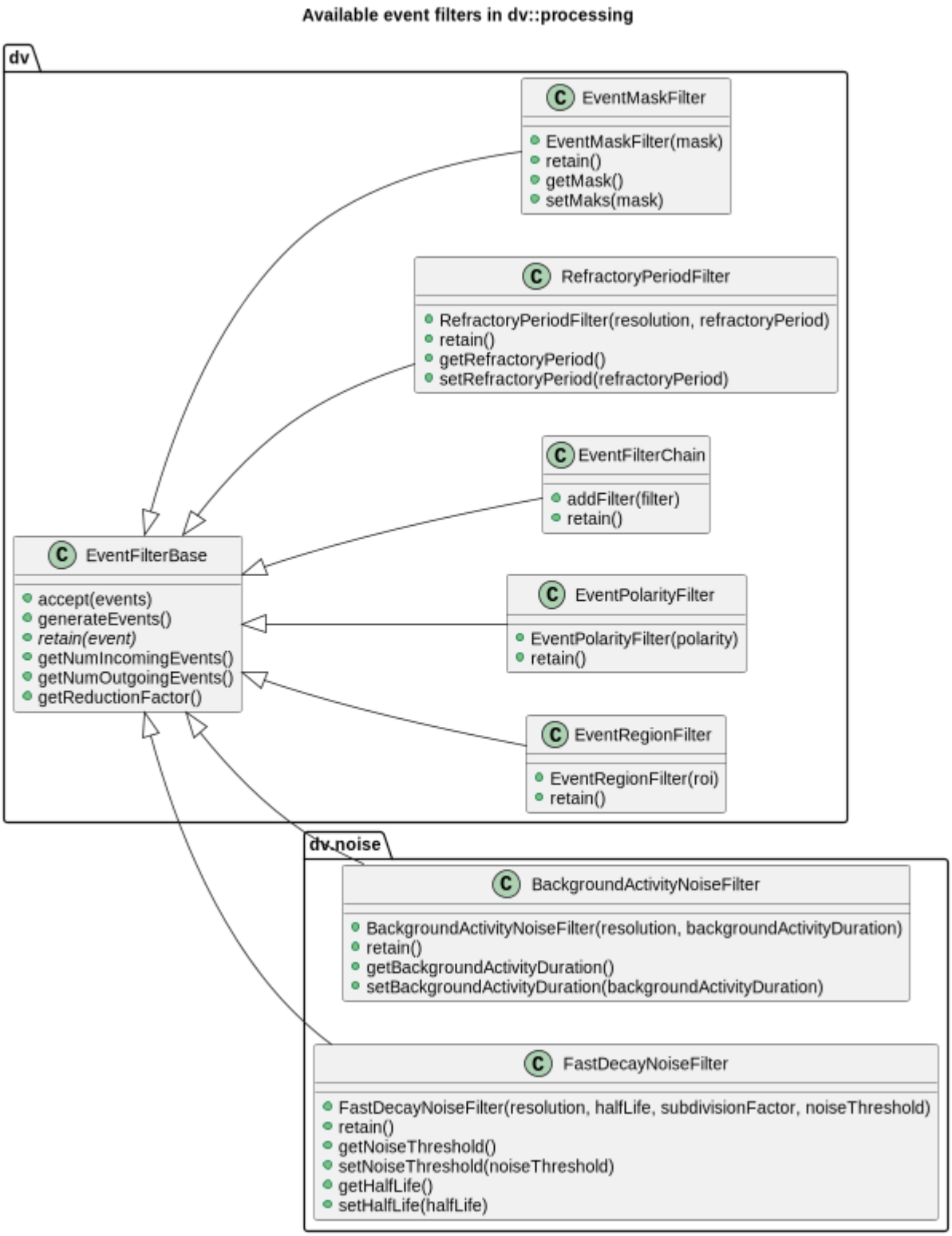


Fig. 8: Available filters and their hierarchy in the library.

(continued from previous page)

```
↪resolution, 1000);

    // Adding additional data for drawing, this will give an idea whether the filter␣
↪removes actual signal events
    events.add(dv::data::generate::dvLogoAsEvents(10000, resolution));

    // Initialize a background activity noise filter with 1-millisecond activity␣
↪period
    dv::noise::BackgroundActivityNoiseFilter filter(resolution, 1ms);

    // Pass events to the filter
    filter.accept(events);

    // Call generate events to apply the noise filter
    const dv::EventStore filtered = filter.generateEvents();

    // Print out the reduction factor, which indicates the percentage of discarded␣
↪events
    std::cout << "Filter reduced number of events by a factor of " << filter.
↪getReductionFactor() << std::endl;

    // Use a visualizer instance to preview the events
    dv::visualization::EventVisualizer visualizer(resolution);

    // Generate preview images of data input and output
    const cv::Mat input  = visualizer.generateImage(events);
    const cv::Mat output = visualizer.generateImage(filtered);

    // Concatenate the images into a single image for preview
    cv::Mat preview;
    cv::hconcat(input, output, preview);

    // Display the input and output images
    cv::namedWindow("preview", cv::WINDOW_NORMAL);
    cv::imshow("preview", preview);
    cv::waitKey();

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Hardcoded VGA resolution
resolution = (640, 480)

# Initializing input events with uniformly distributed events which represent noise
events = dv.data.generate.uniformEventsWithinTimeRange(0, timedelta(milliseconds=10),␣
↪resolution, 1000)

# Adding additional data for drawing, this will give an idea whether the filter␣
↪removes actual signal events
events.add(dv.data.generate.dvLogoAsEvents(10000, resolution))

# Initialize a background activity noise filter with 1-millisecond activity period
```

(continues on next page)

(continued from previous page)

```
filter = dv.noise.BackgroundActivityNoiseFilter(resolution,␣
↪backgroundActivityDuration=timedelta(milliseconds=1))

# Pass events to the filter
filter.accept(events)

# Call generate events to apply the noise filter
filtered = filter.generateEvents()

# Print out the reduction factor, which indicates the percentage of discarded events
print(f"Filter reduced number of events by a factor of {filter.getReductionFactor()}")

# Use a visualizer instance to preview the events
visualizer = dv.visualization.EventVisualizer(resolution)

# Generate preview images of data input and output
input = visualizer.generateImage(events)
output = visualizer.generateImage(filtered)

# Concatenate the images into a single image for preview
preview = cv.hconcat([input, output])

# Display the input and output images
cv.namedWindow("preview", cv.WINDOW_NORMAL)
cv.imshow("preview", preview)
cv.waitKey()
```

The sample code outputs such images:

Fig. 9: Output of the sample use of background activity filter. Left is a preview of input events, right is a preview of filtered events.

Note: The image above that the filter reduces the amount of speckles on white area, but maintains the logo preview, those events are not filtered out.

Fast decay noise filter

dv::noise::FastDecayNoiseFilter - events are filtered based on lower resolution fast-decaying representation of events. Events contribute to a low-resolution accumulated image with a fast decay, which also represents local activity. Unlike the `BackgroundActivityNoiseFilter`, this filter uses decay instead of a hard time threshold, although the approach is very similar - an event needs to be supported by another event in a local pixel neighborhood. This filter has a lower memory footprint since the neighborhood is represented in a low resolution accumulated image.

Following sample code shows the usage of the *dv::noise::FastDecayNoiseFilter* to filter noise:

C++

Python

```cpp
#include <dv-processing/data/generate.hpp>
#include <dv-processing/noise/fast_decay_noise_filter.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Hardcoded VGA resolution
    const cv::Size resolution(640, 480);

    // Initializing input events with uniformly distributed events which represent noise
    dv::EventStore events = dv::data::generate::uniformEventsWithinTimeRange(0, 10ms, resolution, 1000);

    // Adding additional data for drawing, this will give an idea whether the filter removes actual signal events
    events.add(dv::data::generate::dvLogoAsEvents(10000, resolution));

    // Initialize a background activity noise filter with 10-millisecond half life decay, resolution subdivision
    // factor of 4 and noise threshold of 1. Half life decay and noise threshold values controls the quality of
    // filtering, while subdivision factor is used for resolution downsizing for internal event representation.
    dv::noise::FastDecayNoiseFilter filter(resolution, 10ms, 4, 1.f);

    // Pass events to the filter
    filter.accept(events);

    // Call generate events to apply the noise filter
    const dv::EventStore filtered = filter.generateEvents();

    // Print out the reduction factor, which indicates the percentage of discarded events
    std::cout << "Filter reduced number of events by a factor of " << filter.getReductionFactor() << std::endl;

    // Use a visualizer instance to preview the events
    dv::visualization::EventVisualizer visualizer(resolution);

    // Generate preview images of data input and output
    const cv::Mat input  = visualizer.generateImage(events);
```

(continues on next page)

(continued from previous page)

```cpp
    const cv::Mat output = visualizer.generateImage(filtered);

    // Concatenate the images into a single image for preview
    cv::Mat preview;
    cv::hconcat(input, output, preview);

    // Display the input and output images
    cv::namedWindow("preview", cv::WINDOW_NORMAL);
    cv::imshow("preview", preview);
    cv::waitKey();

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Hardcoded VGA resolution
resolution = (640, 480)

# Initializing input events with uniformly distributed events which represent noise
events = dv.data.generate.uniformEventsWithinTimeRange(0, timedelta(milliseconds=10), resolution, 1000)

# Adding additional data for drawing, this will give an idea whether the filter removes actual signal events
events.add(dv.data.generate.dvLogoAsEvents(10000, resolution))

# Initialize a background activity noise filter with 10-millisecond half life decay, resolution subdivision
# factor of 4 and noise threshold of 1. Half life decay and noise threshold values controls the quality of
# filtering, while subdivision factor is used for resolution downsizing for internal event representation.
filter = dv.noise.FastDecayNoiseFilter(resolution,
                                       halfLife=timedelta(milliseconds=10),
                                       subdivisionFactor=4,
                                       noiseThreshold=1.0)

# Pass events to the filter
filter.accept(events)

# Call generate events to apply the noise filter
filtered = filter.generateEvents()

# Print out the reduction factor, which indicates the percentage of discarded events
print(f"Filter reduced number of events by a factor of {filter.getReductionFactor()}")

# Use a visualizer instance to preview the events
visualizer = dv.visualization.EventVisualizer(resolution)

# Generate preview images of data input and output
input = visualizer.generateImage(events)
output = visualizer.generateImage(filtered)

```

(continues on next page)

(continued from previous page)

```
# Concatenate the images into a single image for preview
preview = cv.hconcat([input, output])

# Display the input and output images
cv.namedWindow("preview", cv.WINDOW_NORMAL)
cv.imshow("preview", preview)
cv.waitKey()
```

The sample code outputs such images:

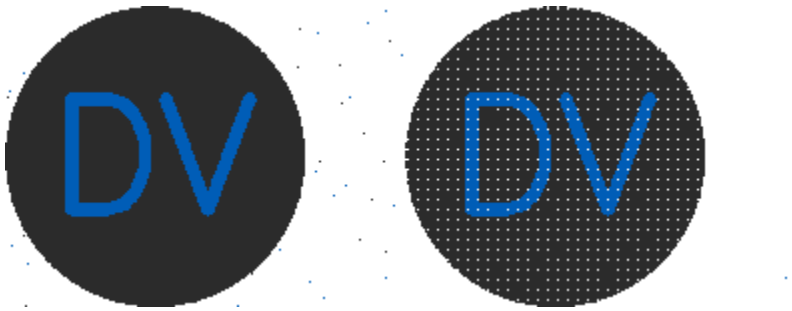


Fig. 10: Output of the sample use of fast decay noise filter. Left is a preview of input events, right is a preview of filtered events.

Note: The image above that the filter drastically the amount of speckles on white area, although the logo image is also affected and some true-signal events contributing to the logo are also filtered out.

3.4.3 Event subsampling

The same filtering approach is used to subsampling events based on their pixel location, polarity or other properties. This chapter describes the available event subsampling filters in the library.

Mask filter

dv::EventMaskFilter - filters events based on a pixel mask. Events are discarded in pixel locations where mask has zero values.

Following sample code shows the usage of the *dv::EventMaskFilter* to filter out selected regions of events:

C++

Python

```
#include <dv-processing/core/filters.hpp>
#include <dv-processing/data/generate.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

```

(continues on next page)

(continued from previous page)

```cpp
    // Smaller resolution for previews
    const cv::Size resolution(200, 200);

    // Initializing input events with events that represent a logo
    const dv::EventStore events = dv::data::generate::dvLogoAsEvents(0, resolution);

    // Initialize a mask with all zero values
    cv::Mat mask(resolution, CV_8UC1, cv::Scalar(0));

    // Draw two rectangles to generate a similar to checkerboard mask pattern
    cv::rectangle(
        mask, cv::Point(0, 0), cv::Point(resolution.width / 2, resolution.height / 2),
↪ cv::Scalar(255), cv::FILLED);
    cv::rectangle(mask, cv::Point(resolution.width / 2, resolution.height / 2),
        cv::Point(resolution.width, resolution.height), cv::Scalar(255), cv::FILLED);

    // Initialize the mask filter with the generated mask
    dv::EventMaskFilter filter(mask);

    // Pass events to the filter
    filter.accept(events);

    // Call generate events to apply the filter
    const dv::EventStore filtered = filter.generateEvents();

    // Print out the reduction factor, which indicates the percentage of discarded␣
↪events
    std::cout << "Filter reduced number of events by a factor of " << filter.
↪getReductionFactor() << std::endl;

    // Use a visualizer instance to preview the events
    dv::visualization::EventVisualizer visualizer(resolution);

    // Generate preview images of data input and output
    const cv::Mat input  = visualizer.generateImage(events);
    const cv::Mat output = visualizer.generateImage(filtered);

    // Concatenate the images into a single image for preview
    cv::Mat preview, maskColored;
    cv::cvtColor(mask, maskColored, cv::COLOR_GRAY2BGR);
    cv::hconcat(std::vector<cv::Mat>({input, maskColored, output}), preview);

    // Display the input and output images
    cv::namedWindow("preview", cv::WINDOW_NORMAL);
    cv::imshow("preview", preview);
    cv::waitKey();

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
import numpy as np

# Smaller resolution for previews
resolution = (200, 200)
```

(continues on next page)

(continued from previous page)

```

# Initializing input events with events that represent a logo
events = dv.data.generate.dvLogoAsEvents(0, resolution)

# Initialize a mask with all zero values
mask = np.full(resolution, fill_value=0, dtype=np.uint8)

# Draw two rectangles to generate a similar to checkerboard mask pattern
cv.rectangle(mask, [0, 0], [int(resolution[0] / 2), int(resolution[1] / 2)], (255, ),␣
↪cv.FILLED)
cv.rectangle(mask, [int(resolution[0] / 2), int(resolution[1] / 2)], resolution, (255,
↪ ), cv.FILLED)

# Initialize the mask filter with the generated mask
filter = dv.EventMaskFilter(mask)

# Pass events to the filter
filter.accept(events)

# Call generate events to apply the filter
filtered = filter.generateEvents()

# Print out the reduction factor, which indicates the percentage of discarded events
print(f"Filter reduced number of events by a factor of {filter.getReductionFactor()}")

# Use a visualizer instance to preview the events
visualizer = dv.visualization.EventVisualizer(resolution)

# Generate preview images of data input and output
input = visualizer.generateImage(events)
output = visualizer.generateImage(filtered)

# Concatenate the images into a single image for preview
preview = cv.hconcat([input, cv.cvtColor(mask, cv.COLOR_GRAY2BGR), output])

# Display the input and output images
cv.namedWindow("preview", cv.WINDOW_NORMAL)
cv.imshow("preview", preview)
cv.waitKey()
```

The sample code outputs such images:

Fig. 11: Output of the sample use of event mask filter. Left is a preview of input events, middle is the mask used, and right is a preview of filtered events.

Refractory period filter

dv::RefractoryPeriodFilter - refractory period filter discards bursts of events at repeating pixel locations. Each event timestamp is compared against most recent event timestamp on the same pixel location, if the timestamp difference is less than the refractory period, the event is discarded.

Following sample code shows the usage of the *dv::RefractoryPeriodFilter* to filter out events that are within close time period on the same coordinate location:

C++

Python

```cpp
#include <dv-processing/core/filters.hpp>
#include <dv-processing/data/generate.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    const cv::Size resolution(200, 200);

    // Initializing 10000 events that are uniformly spaced in pixel area and time
    dv::EventStore events = dv::data::generate::uniformEventsWithinTimeRange(0, 10ms, resolution, 10000);

    // Initialize refractory period filter with 1-millisecond period
    dv::RefractoryPeriodFilter filter(resolution, 1ms);

    // Pass events to the filter
    filter.accept(events);

    // Call generate events to apply the filter
    const dv::EventStore filtered = filter.generateEvents();

    // Print out the number of events after filtering
    std::cout << "Filtered [" << filtered.size() << "] events out of [" << events.size() << "]" << std::endl;

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

resolution = (200, 200)

# Initializing 10000 events that are uniformly spaced in pixel area and time
events = dv.data.generate.uniformEventsWithinTimeRange(0, timedelta(milliseconds=10), resolution, 10000)

# Initialize refractory period filter with 1-millisecond period
filter = dv.RefractoryPeriodFilter(resolution, timedelta(milliseconds=1))

# Pass events to the filter
filter.accept(events)
```

(continues on next page)

(continued from previous page)

```
15
16 # Call generate events to apply the filter
17 filtered = filter.generateEvents()
18
19 # Print out the number of events after filtering
20 print(f"Filtered [{len(filtered)}] events ouf of [{len(events)}]")
```

Polarity filter

dv::EventPolarityFilter - filter events based on polarity.

Following sample code shows the usage of the *dv::EventPolarityFilter* to filter out events based on polarity:

C++

Python

```
1  #include <dv-processing/core/filters.hpp>
2  #include <dv-processing/data/generate.hpp>
3  #include <dv-processing/visualization/event_visualizer.hpp>
4
5  #include <opencv2/highgui.hpp>
6
7  int main() {
8      using namespace std::chrono_literals;
9
10     const cv::Size resolution(200, 200);
11
12     // Initializing input events with events that represent a logo
13     dv::EventStore events = dv::data::generate::dvLogoAsEvents(0, resolution);
14
15     // Filter positive polarity events only
16     dv::EventPolarityFilter filter(true);
17
18     // Pass events to the filter
19     filter.accept(events);
20
21     // Call generate events to apply the filter
22     const dv::EventStore filtered = filter.generateEvents();
23
24     // Use a visualizer instance to preview the events
25     dv::visualization::EventVisualizer visualizer(resolution);
26
27     // Generate preview images of data input and output
28     const cv::Mat input  = visualizer.generateImage(events);
29     const cv::Mat output = visualizer.generateImage(filtered);
30
31     // Concatenate the images into a single image for preview
32     cv::Mat preview;
33     cv::hconcat(input, output, preview);
34
35     // Display the input and output images
36     cv::namedWindow("preview", cv::WINDOW_NORMAL);
37     cv::imshow("preview", preview);
38     cv::waitKey();
39
```

(continues on next page)

(continued from previous page)

```
    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

resolution = (200, 200)

# Initializing input events with events that represent a logo
events = dv.data.generate.dvLogoAsEvents(0, resolution)

# Filter positive polarity events only
filter = dv.EventPolarityFilter(True)

# Pass events to the filter
filter.accept(events)

# Call generate events to apply the filter
filtered = filter.generateEvents()

# Use a visualizer instance to preview the events
visualizer = dv.visualization.EventVisualizer(resolution)

# Generate preview images of data input and output
input = visualizer.generateImage(events)
output = visualizer.generateImage(filtered)

# Concatenate the images into a single image for preview
preview = cv.hconcat([input, output])

# Display the input and output images
cv.namedWindow("preview", cv.WINDOW_NORMAL)
cv.imshow("preview", preview)
cv.waitKey()
```

The sample code outputs such images:

Fig. 12: Output of the sample use of polarity filter. Left is a preview of input events and right is a preview of filtered events.

Note: The generated DV logo image comes from event representation, white area means no events are there, dark grey area are negative events and pixels are blue on coordinates where positive polarity events are provided. The filtered image only contains the letters "DV" since the background circle is represented by negative polarity events.

Event region filter

dv::EventRegionFilter - filter events based on given region of interest.

Following sample code shows the usage of the *dv::EventRegionFilter* to filter out specific area of events:

C++

Python

```cpp
#include <dv-processing/core/filters.hpp>
#include <dv-processing/data/generate.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    const cv::Size resolution(200, 200);

    // Initializing input events with events that represent a logo
    dv::EventStore events = dv::data::generate::dvLogoAsEvents(0, resolution);

    // Initialize region filter using hardcoded coordinates
    dv::EventRegionFilter filter(cv::Rect(50, 50, 100, 100));

    // Pass events to the filter
    filter.accept(events);

    // Call generate events to apply the filter
    const dv::EventStore filtered = filter.generateEvents();

    // Use a visualizer instance to preview the events
    dv::visualization::EventVisualizer visualizer(resolution);

    // Generate preview images of data input and output
    const cv::Mat input  = visualizer.generateImage(events);
    const cv::Mat output = visualizer.generateImage(filtered);

    // Concatenate the images into a single image for preview
    cv::Mat preview;
    cv::hconcat(input, output, preview);

    // Display the input and output images
    cv::namedWindow("preview", cv::WINDOW_NORMAL);
    cv::imshow("preview", preview);
    cv::waitKey();

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

resolution = (200, 200)

# Initializing input events with events that represent a logo
events = dv.data.generate.dvLogoAsEvents(0, resolution)

# Initialize region filter using hardcoded coordinates
filter = dv.EventRegionFilter((50, 50, 100, 100))

# Pass events to the filter
filter.accept(events)

# Call generate events to apply the filter
filtered = filter.generateEvents()

# Use a visualizer instance to preview the events
visualizer = dv.visualization.EventVisualizer(resolution)

# Generate preview images of data input and output
input = visualizer.generateImage(events)
output = visualizer.generateImage(filtered)

# Concatenate the images into a single image for preview
preview = cv.hconcat([input, output])

# Display the input and output images
cv.namedWindow("preview", cv.WINDOW_NORMAL)
cv.imshow("preview", preview)
cv.waitKey()
```

The sample code outputs such images:

Fig. 13: Output of the sample use of region filter. Left is a preview of input events and right is a preview of filtered events.

Filter chain

Multiple filters can be combined into a single filter chain, which optimizes memory operations to increase the performance of applying multiple filters. This is achieved by using the *dv::EventFilterChain* class. Multiple filters can be added using *dv::EventFilterChain::addFilter()* method, it accepts filter wrapped in `std::shared_pointer`, the shared pointer is used to be able to modify the parameters of filters after they are added to the filter chain.

Following sample code shows the usage of the *dv::EventFilterChain* to apply multiple types of filters in a single chain:

C++

Python

```cpp
#include <dv-processing/core/filters.hpp>
#include <dv-processing/data/generate.hpp>
#include <dv-processing/noise/background_activity_noise_filter.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    const cv::Size resolution(200, 200);

    // Initializing input events with events that represent a logo
    dv::EventStore events = dv::data::generate::dvLogoAsEvents(0, resolution);

    // Initialize event filter chain, it contains no filters
    dv::EventFilterChain filter;

    // Now let's add filters
    // First, add a region filter with hardcoded coordinates
    filter.addFilter(std::make_shared<dv::EventRegionFilter<>>(cv::Rect(50, 50, 100,␣
↪100)));

    // Second, add a positive polarity filter
    filter.addFilter(std::make_shared<dv::EventPolarityFilter<>>(true));

    // Third, add a background activity noise filter
    filter.addFilter(std::make_shared<dv::noise::BackgroundActivityNoiseFilter<>>
↪(resolution));

    // Pass events to the filter
    filter.accept(events);

    // Call generate events to apply the filter chain, it will apply all three filters
    const dv::EventStore filtered = filter.generateEvents();

    // Use a visualizer instance to preview the events
    dv::visualization::EventVisualizer visualizer(resolution);

    // Generate preview images of data input and output
    const cv::Mat input  = visualizer.generateImage(events);
    const cv::Mat output = visualizer.generateImage(filtered);

```

(continues on next page)

(continued from previous page)

```cpp
    // Concatenate the images into a single image for preview
    cv::Mat preview;
    cv::hconcat(input, output, preview);

    // Display the input and output images
    cv::namedWindow("preview", cv::WINDOW_NORMAL);
    cv::imshow("preview", preview);
    cv::waitKey();

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

resolution = (200, 200)

# Initializing input events with events that represent a logo
events = dv.data.generate.dvLogoAsEvents(0, resolution)

# Initialize event filter chain, it contains no filters
filter = dv.EventFilterChain()

# Now let's add filters
# First, add a region filter with hardcoded coordinates
filter.addFilter(dv.EventRegionFilter((50, 50, 100, 100)))

# Second, add a positive polarity filter
filter.addFilter(dv.EventPolarityFilter(True))

# Third, add a background activity noise filter
filter.addFilter(dv.noise.BackgroundActivityNoiseFilter(resolution))

# Pass events to the filter
filter.accept(events)

# Call generate events to apply the filter
filtered = filter.generateEvents()

# Use a visualizer instance to preview the events
visualizer = dv.visualization.EventVisualizer(resolution)

# Generate preview images of data input and output
input = visualizer.generateImage(events)
output = visualizer.generateImage(filtered)

# Concatenate the images into a single image for preview
preview = cv.hconcat([input, output])

# Display the input and output images
cv.namedWindow("preview", cv.WINDOW_NORMAL)
cv.imshow("preview", preview)
cv.waitKey()
```

The sample code outputs such images:

Fig. 14: Output of the sample use of multiple filters in a filter chain. Left is a preview of input events and right is a preview of filtered events.

3.4.4 Filtering performance

The provided event filters performance is measured using benchmarks, the benchmarks for filters can be found under directory `benchmarks/noise` in the project repository. These are sample benchmarking results, performance of filters is measured in throughput of mega-events per second. These measurements were capture on an AMD Ryzen 5 3600 6-Core processor.

Filter	Event count per iteration	Throughput, MegaEvents / second
FastDecayNoiseFilter	1000	82.3
FastDecayNoiseFilter	4096	82.9
FastDecayNoiseFilter	32768	80.1
FastDecayNoiseFilter	262144	67.5
FastDecayNoiseFilter	1000000	67.2
BackgroundActivityNoiseFilter	1000	141.5
BackgroundActivityNoiseFilter	4096	139.5
BackgroundActivityNoiseFilter	32768	105.4
BackgroundActivityNoiseFilter	262144	134.6
BackgroundActivityNoiseFilter	1000000	135.0
RefractoryPeriodFilter	1000	268.7
RefractoryPeriodFilter	4096	278.9
RefractoryPeriodFilter	32768	254.9
RefractoryPeriodFilter	262144	255.1
RefractoryPeriodFilter	1000000	167.5
PolarityFilter	1000	503.7
PolarityFilter	4096	345.0
PolarityFilter	32768	165.4
PolarityFilter	262144	157.7
PolarityFilter	1000000	156.6
RegionFilter	1000	446.9
RegionFilter	4096	371.7
RegionFilter	32768	166.8
RegionFilter	262144	153.6
RegionFilter	1000000	151.0
MaskFilter	1000	421.5
MaskFilter	4096	264.8

continues on next page

Table 1 – continued from previous page

Filter	Event count per iteration	Throughput, MegaEvents / second
MaskFilter	32768	123.6
MaskFilter	262144	118.0
MaskFilter	1000000	117.1
ThreeFiltersNoChain	1000	89.3
ThreeFiltersNoChain	4096	88.0
ThreeFiltersNoChain	32768	76.2
ThreeFiltersNoChain	262144	75.0
ThreeFiltersNoChain	1000000	72.6
ThreeFiltersWithChain	1000	170.4
ThreeFiltersWithChain	4096	99.1
ThreeFiltersWithChain	32768	69.4
ThreeFiltersWithChain	262144	70.3
ThreeFiltersWithChain	1000000	70.1

3.5 Command-line utilities

The dv-processing library provides certain command-line utilities that are useful for troubleshooting basic issues or performing some basic tasks.

3.5.1 dv-filestat

The `dv-filestat` utility is useful for inspection of AEDAT4 files. It provides information on the content of a given file. Usage:

```
$ dv-filestat path/to/file.aedat4
```

Sample output from this utility:

```
$ dv-filestat ~/dvSave-2022_06_29_13_40_44.aedat4
File path (canonical): "/home/rokas/dvSave-2022_06_29_13_40_44.aedat4"
File size (OS): 25429285
File size (Parser): 25429285
Compression: LZ4
Timestamp lowest: 1656502844057775
Timestamp highest: 1656502852007717
Timestamp difference: 7949942
Timestamp shift: 1656502844057775
Stream 0: events - EVTS
Stream 2: imu - IMUS
Stream 3: triggers - TRIG
DataTable file position: 25377171
DataTable file size: 52114
DataTable elements: 1590
```

Here, the provided fields are:

- File path - absolute file path in the filesystem;
- File size - actual file size in bytes;
- Compression - compression type;

- Timestamp - timing information in Unix microsecond format;
- Stream - available streams (e.g. "Stream 0: events - EVTS"), where "0" - stream id, "events" - stream name, "EVTS" - stream type identifier;
- DataTable - internal data layout information.

It is possible to pass a verbose flag ("-v" or "–verbose") to obtain more information on the recorded data. Sample verbose output:

```
$ dv-filestat -v ~/dvSave-2022_06_29_13_40_44.aedat4
File path (canonical): "/home/rokas/dvSave-2022_06_29_13_40_44.aedat4"
File size (OS): 25429285
File size (Parser): 25429285
Compression: LZ4
Timestamp lowest: 1656502844057775
Timestamp highest: 1656502852007717
Timestamp difference: 7949942
Timestamp shift: 1656502844057775
Stream 0: events - EVTS
XML content:
0/
compression = LZ4
originalModuleName = capture
originalOutputName = events
typeDescription = Array of events (polarity ON/OFF).
typeIdentifier = EVTS
  0/info/
  sizeX = 640
  sizeY = 480
  source = DVXplorer_DXA00093
  tsOffset = 1656502833725006

Stream 2: imu - IMUS
XML content:
2/
compression = LZ4
originalModuleName = capture
originalOutputName = imu
typeDescription = Inertial Measurement Unit data samples.
typeIdentifier = IMUS
  2/info/
  source = DVXplorer_DXA00093
  tsOffset = 1656502833725006

Stream 3: triggers - TRIG
XML content:
3/
compression = LZ4
originalModuleName = capture
originalOutputName = triggers
typeDescription = External triggers and special signals.
typeIdentifier = TRIG
  3/info/
  source = DVXplorer_DXA00093
  tsOffset = 1656502833725006

DataTable file position: 25377171
DataTable file size: 52114
```

(continues on next page)

(continued from previous page)

```
DataTable elements: 1590
Packet at 2342: StreamID 0 - Size 31375 - NumElements 4911 - TimestampStart␣
↪1656502844057775 - TimestampEnd 1656502844067773
Packet at 33725: StreamID 2 - Size 401 - NumElements 8 - TimestampStart␣
↪1656502844057878 - TimestampEnd 1656502844066774
Packet at 34134: StreamID 0 - Size 30591 - NumElements 4815 - TimestampStart␣
↪1656502844067783 - TimestampEnd 1656502844077744
Packet at 64733: StreamID 2 - Size 374 - NumElements 8 - TimestampStart␣
↪1656502844068045 - TimestampEnd 1656502844076941
Packet at 65115: StreamID 0 - Size 32736 - NumElements 5131 - TimestampStart␣
↪1656502844077777 - TimestampEnd 1656502844087761
Packet at 97859: StreamID 2 - Size 378 - NumElements 8 - TimestampStart␣
↪1656502844078212 - TimestampEnd 1656502844087107
Packet at 98245: StreamID 0 - Size 31442 - NumElements 4933 - TimestampStart␣
↪1656502844087779 - TimestampEnd 1656502844097756
Packet at 129695: StreamID 2 - Size 378 - NumElements 8 - TimestampStart␣
↪1656502844088378 - TimestampEnd 1656502844097274
```

The verbose output now prints each streams metadata, e.g.:

```
Stream 0: events - EVTS
XML content:
0/
compression = LZ4
originalModuleName = capture
originalOutputName = events
typeDescription = Array of events (polarity ON/OFF).
typeIdentifier = EVTS
  0/info/
  sizeX = 640
  sizeY = 480
  source = DVXplorer_DXA00093
  tsOffset = 1656502833725006
```

The XML metadata provide information on the compression type, some information on the type, as well as source camera name and the resolution of the data.

Additionally, packet information is printed, e.g.:

```
Packet at 2342: StreamID 0 - Size 31375 - NumElements 4911 - TimestampStart␣
↪1656502844057775 - TimestampEnd 1656502844067773
```

The information here represents:

- “Packet at 2342” - the 2342 is the byte start index of this packet in the file;
- “StreamID 0” - id of the stream, which matches stream ids from output above;
- “Size 31375” - size of the content of this packet in bytes;
- “NumElements 4911” - number of element in packet, since it’s an event packet, it contains 4911 events;
- “TimestampStart 1656502844057775 - TimestampEnd 1656502844067773” - this is the timing information of this packet, it contains data from time 1656502844057775 to 1656502844067773. These are Unix microsecond timestamps.

3.5.2 dv-list-devices

The `dv-list-devices` is a basic utility that allows inspection of connected iniVation cameras on the system. Sample output from this command:

```
$ dv-list-devices
Device discovery: found 1 devices.
Detected device [DVXplorer_DXA00093]
```

Running the command without any additional flags will just print the list of available cameras by their names. By adding a verbose flag ("-v" or "–verbose"), it will print additional information about the available cameras, a sample output:

```
$ dv-list-devices --verbose
Device discovery: found 1 devices.
Detected device [DVXplorer_DXA00093]
- DVXplorer (type 8)
        - USB busNum:devAddr: 6:14
        - Device can be opened: true
        - USB serial number: DXA00093
        - Device needs firmware update: false
        - Timestamp Master: true
        - Firmware Version: 8
        - Logic Version: 18
        - Chip ID: 20
        - DVS Size X: 640
        - DVS Size Y: 480
        - DVS Statistics: true
        - External IO Generator: false
        - Multiplexer Statistics: true
        - IMU Model: Bosch BMI160
```

3.5.3 dv-imu-bias-estimation

The `dv-imu-bias-estimation` tool is used to estimate intrinsic measurement biases that are measured by the IMU device when there is no motion to be measured. The biases are internal offsets in the measuring device that come from imperfections during manufacturing process of the IMU device.

Usage:

```
Usage: dv-imu-bias-estimation [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  -c,--camera-name TEXT       Provide camera name to open. Application will open␣
↪first discovered camera if a name is not provided.
  -t,--variance-threshold FLOAT [0.1]
                              Maximum variance that can be measured on IMU data. This␣
↪value is used to determine motion in the data.
  -d,--duration FLOAT [1]     Time duration for collecting sample data in seconds.
  -a,--calibration-file TEXT:FILE
                              Path to the calibration file to store the biases values.
```

To estimate the biases, first place the camera steady on a level surface and run the utility:

```
$ dv-imu-bias-estimation
Opened camera [DVXplorer_DXA00093]
```

(continues on next page)

(continued from previous page)

```
Keep the camera steady on a level surface and press ENTER to start data collection ...

Collected 794 samples over 1.007546 seconds
Bias estimation successful!
Accelerometer biases [x, y, z] in m/s^2:
        [-0.4947727, -0.7454767, 0.12935007]
Gyroscope biases [x, y, z] in rad/s:
        [0.0028670905, 0.004207727, 0.0023824223]
```

Fig. 15: Camera placed on a level table surface for imu bias estimation.

When running without any parameters, the utility will open the first discovered camera and will perform bias estiamtion for it. It will prompt the user to place the camera on a level surface and press enter key to start collection. When the key is pressed, it will collect 1 second of IMU measurements and estimate the biases which are printed into the screen. These values can be subtracted from raw measurements of IMU to compensate for the intrinsic offsets the IMU is measuring.

The utility also provides additional parameters, below is a detailed explanation of these value:

- `-c,--camera-name` - provide a specific camera name to open. By default, the application will open first discovered device on the system, by providing a specific name, it will open the specified device.
- `-t,--variance-threshold` - to ensure that the camera is stationary while collecting data, the utility is measuring variance on the measured data. It will throw an exception if variance of at least one measurement is

exceeding the given threshold. If this threshold is exceeded and the camera was not moved, this value can be increased to bypass this check.

- `-d,--duration` - duration of data collection in seconds. Usually one second is sufficient for a general case. The duration value can be tuned if estimated biases are inconsistent.
- `-a,--calibration-file` - the estimated bias values are only printed to the terminal output. It is also possible to save them in a calibration file for persistence. The utility will open the given file (the file needs to exist and contain a calibration) and update the IMU parameter values. If an IMU calibration for the camera already exists, it will overwrite previous settings, if IMU calibration for this camera does not exist, the calibration will be added.

3.5.4 dv-tcpstat

The `dv-tcpstat` utility is useful for inspection of remote TCP server streams. It provides information on the stream the remote server provides. It is compatible with TCP remote streaming instances created using *`dv::io::NetworkWriter`* class and with DV's Output net tcp server module[7].

Usage:

```
$ dv-tcpstat
Connect to a TCP streaming server and print information about it
Usage: dv-tcpstat [OPTIONS]

Options:
  -h,--help                   Print this help message and exit
  -v,--verbose                Print full packet table
  -r,--try-reading            Try reading and printing packet information
  -i,--ip TEXT:IPV4 REQUIRED  IP address of the server to connect to
  -p,--port UINT:INT in [0 - 65535] REQUIRED
                              Server port number
```

As a sample demonstration of usage, let's take a DV project that outputs event stream into the output TCP module.

The module is configured to listen on IP address `127.0.0.1` and port `46581`, we can inspect this server using the `dv-tcpstat` utility from the same computer. When running the utility with given IP, port, and verbose output flag, it prints the information about the available data:

```
$ dv-tcpstat -i 127.0.0.1 -p 46581 -v
Attempting to connect to [127.0.0.1:46581]...
Connected to [127.0.0.1:46581]!

Stream info on stream ID 0:
        Stream name: "tcpserver"
        Stream type identifier: "EVTS"
        Stream details:
                /0/
                originalOutputName = tcpserver
                typeDescription = Array of events (polarity ON/OFF).
                typeIdentifier = EVTS
                                /0/info/
                                sizeX = 640
                                sizeY = 480
                                source = DVXplorer_DXA00093
```

The executable will print this information and disconnect immediately. It is possible to try and read some data from the remote server by passing the `-r,--try-reading` flag, this is the expected with this option enabled:

[7] https://docs.inivation.com/software/dv/modules/built-in-modules/output-network.html#output-tcp-module

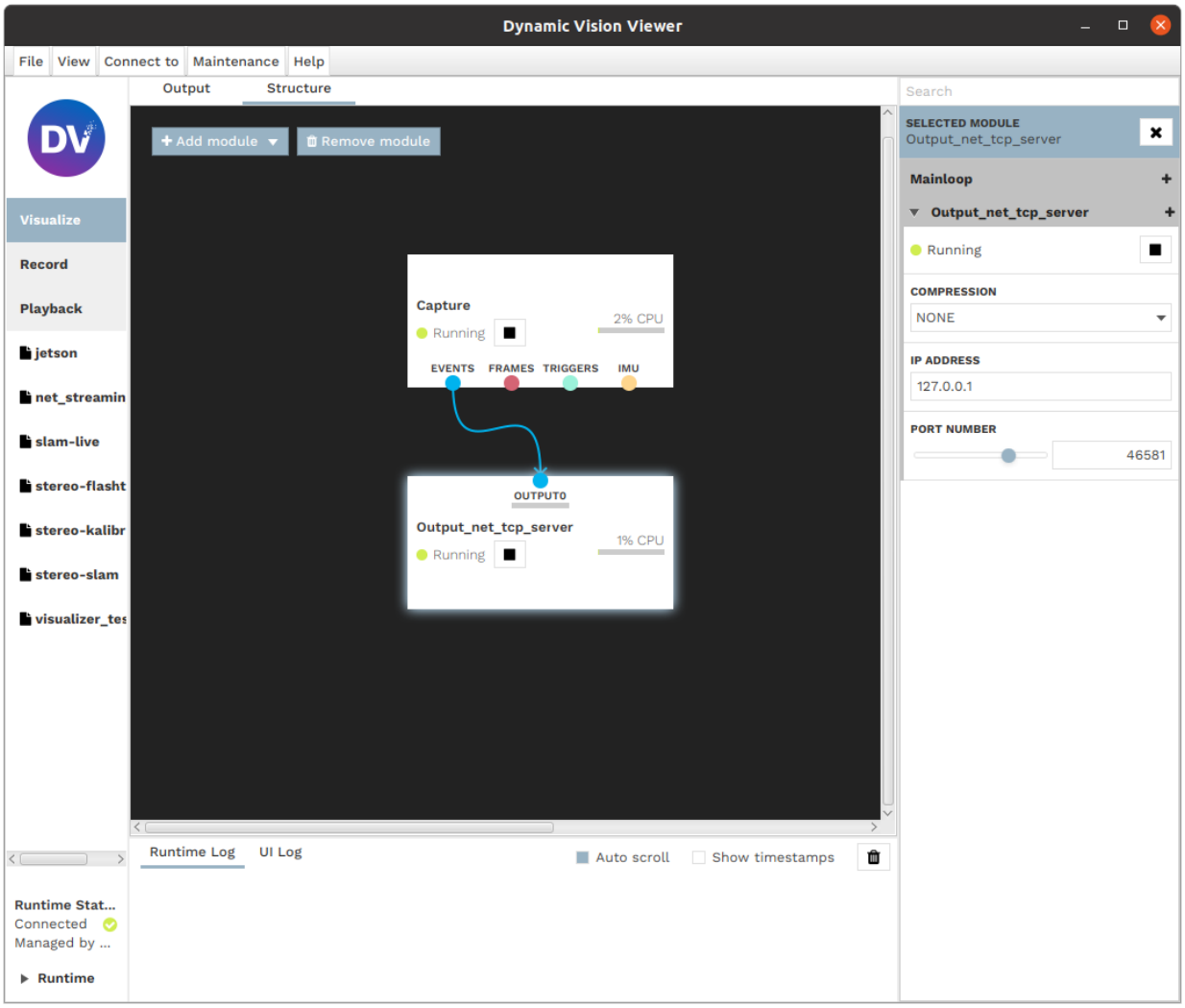


Fig. 16: A project in DV running a live camera connected to a TCP output server module.

```
$ dv-tcpstat -i 127.0.0.1 -p 46581 -r
Attempting to connect to [127.0.0.1:46581]...
Connected to [127.0.0.1:46581]!

Stream info on stream ID 0:
        Stream name: "tcpserver"
        Stream type identifier: "EVTS"
        Stream details:
                /0/
                originalOutputName = tcpserver
                typeDescription = Array of events (polarity ON/OFF).
                typeIdentifier = EVTS
                                /0/info/
                                sizeX = 640
                                sizeY = 480
                                source = DVXplorer_DXA00093
[2022-11-25 09:31:33] Received: EventStore containing 3103 events within 29416µs␣
↪duration; time range within [1669365093787516; 1669365093816932]
[2022-11-25 09:31:33] Received: EventStore containing 3100 events within 29007µs␣
↪duration; time range within [1669365093816932; 1669365093845939]
[2022-11-25 09:31:33] Received: EventStore containing 3105 events within 29034µs␣
↪duration; time range within [1669365093845939; 1669365093874973]
[2022-11-25 09:31:33] Received: EventStore containing 3107 events within 28213µs␣
↪duration; time range within [1669365093874973; 1669365093903186]
[2022-11-25 09:31:33] Received: EventStore containing 3090 events within 29441µs␣
↪duration; time range within [1669365093903186; 1669365093932627]
[2022-11-25 09:31:33] Received: EventStore containing 3097 events within 29614µs␣
↪duration; time range within [1669365093932627; 1669365093962241]
[2022-11-25 09:31:33] Received: EventStore containing 3109 events within 27834µs␣
↪duration; time range within [1669365093962241; 1669365093990075]
[2022-11-25 09:31:33] Received: EventStore containing 3107 events within 27601µs␣
↪duration; time range within [1669365093990075; 1669365094017676]
```

The executable will print packet information until a interrupt signal is sent by the user.

3.6 Reading camera data

The dv-processing library provides a convenient way of reading camera data from live connected cameras and persistent files.

3.6.1 Camera name

Camera name is used to identify a unique camera produced by iniVation. Camera name consists of a camera model and a serial number, concatenated by an underscore ("_") character. The library refers to camera name in multiple methods, this value can be consistently used across the library.

Some examples of a camera name:

- DVXplorer: `DVXplorer_DXA00093`, `DVXplorer_DXM00123`
- DAVIS: `DAVIS346_00000499`

Note: This definition is valid for USB cameras, the camera name is also reported in network streaming sources. In that case, camera name can be manually set by the developer, so naming convention for the models might not be entirely

followed.

3.6.2 From a camera

The easiest approach to access data from a live camera is to use the *dv::io::CameraCapture* class. This section provides in-depth explanation on the usage and code samples.

Discover connected cameras

The camera name can be inspected using a command-line utility *dv-list-devices* that is available in the packages of dv-processing. Sample output for the utility:

```
$ dv-list-devices
Device discovery: found 2 devices.
Detected device [DAVIS346_00000499]
Detected device [DVXplorer_DXA00093]
```

Device discovery is also possible with the use of library methods. Following is a sample on how to detect connected devices using discovery method:

C++

Python

```cpp
#include <dv-processing/io/discovery.hpp>

#include <iostream>

int main() {
    // Call the discovery method
    const std::vector<std::string> cameras = dv::io::discoverDevices();

    std::cout << "Device discovery: found " << cameras.size() << " devices." <<␣
↪std::endl;

    // Loop through detected camera names and print them
    for (const auto &cameraName : cameras) {
        std::cout << "Detected device [" << cameraName << "]" << std::endl;
    }

    return 0;
}
```

```python
import dv_processing as dv

cameras = dv.io.discoverDevices()

print(f"Device discovery: found {len(cameras)} devices.")
for camera_name in cameras:
    print(f"Detected device [{camera_name}]")
```

Opening a camera

The *dv::io::CameraCapture* class follows RAII[8] pattern for resource management. Creating an instance of the class will open the camera connected on USB and starts reading the data immediately, the resources are released when the object instance is destroyed.

The constructor of this class accepts two arguments: camera name [string] and camera type [enum] that are used to specify which camera needs to be opened. The default argument values are designed to not constrain the camera specification and effectively opens first detected camera in the system.

C++

Python

```
#include <dv-processing/io/camera_capture.hpp>

// Open first detected camera in the system
dv::io::CameraCapture capture;
```

```
import dv_processing as dv

capture = dv.io.CameraCapture()
```

It's also possible to open a specific camera on the system, by providing a camera name:

C++

Python

```
// Open the specified camera
dv::io::CameraCapture capture("DVXplorer_DXA000000");
```

```
import dv_processing as dv

# Open the specified camera
capture = dv.io.CameraCapture(cameraName="DVXplorer_DXA000000")
```

Camera type argument can be used to open a camera of given type. If both parameters are provided, the camera will need to match both field requirements to be opened by the *dv::io::CameraCapture* class:

C++

Python

```
// Open any DAVIS camera (camera name not specified)
dv::io::CameraCapture capture("", dv::io::CameraCapture::CameraType::DAVIS);
```

```
import dv_processing as dv

# Open any DAVIS camera (camera name not specified)
capture = dv.io.CameraCapture(type=dv.io.CameraCapture.CameraType.DAVIS)
```

[8] https://en.wikipedia.org/wiki/Resource_acquisition_is_initialization

Checking camera capabilities

The *dv::io::CameraCapture* class abstracts all cameras manufactured by iniVation, since some camera provide different data types, the capture class provides methods to test what data the camera can provide:

C++

Python

```cpp
#include <dv-processing/io/camera_capture.hpp>

int main() {
    // Open any camera
    dv::io::CameraCapture capture;

    // Print the camera name
    std::cout << "Opened [" << capture.getCameraName() << "] camera, it provides:" <<
↪std::endl;

    // Check whether event stream is available
    if (capture.isEventStreamAvailable()) {
        // Get the event stream resolution, the output is a std::optional, so the
↪value() method is
        // used to get the actual resolution value
        const cv::Size resolution = capture.getEventResolution().value();

        // Print the event stream capability with resolution value
        std::cout << "* Events at " << resolution << " resolution" << std::endl;
    }

    // Check whether frame stream is available
    if (capture.isFrameStreamAvailable()) {
        // Get the frame stream resolution
        const cv::Size resolution = capture.getFrameResolution().value();

        // Print the frame stream capability with resolution value
        std::cout << "* Frames at " << resolution << " resolution" << std::endl;
    }

    // Check whether the IMU stream is available
    if (capture.isImuStreamAvailable()) {
        // Print the imu data stream capability
        std::cout << "* IMU measurements" << std::endl;
    }

    // Check whether the trigger stream is available
    if (capture.isTriggerStreamAvailable()) {
        // Print the trigger stream capability
        std::cout << "* Triggers" << std::endl;
    }

    return 0;
}
```

```python
import dv_processing as dv

# Open any camera
capture = dv.io.CameraCapture()
```

(continues on next page)

(continued from previous page)

```python

# Print the camera name
print(f"Opened [{capture.getCameraName()}] camera, it provides:")

# Check whether event stream is available
if capture.isEventStreamAvailable():
    # Get the event stream resolution
    resolution = capture.getEventResolution()

    # Print the event stream capability with resolution value
    print(f"* Events at ({resolution.width}x{resolution.height}) resolution")

# Check whether frame stream is available
if capture.isFrameStreamAvailable():
    # Get the frame stream resolution
    resolution = capture.getFrameResolution()

    # Print the frame stream capability with resolution value
    print(f"* Frames at ({resolution.width}x{resolution.height}) resolution")

# Check whether the IMU stream is available
if capture.isImuStreamAvailable():
    # Print the imu data stream capability
    print("* IMU measurements")

# Check whether the trigger stream is available
if capture.isTriggerStreamAvailable():
    # Print the trigger stream capability
    print("* Triggers")
```

Configuring camera options

Some advanced properties of our cameras can be configured by a number of functions. They are listed here for reference, please check their detailed API documentation for more details.

DVXplorer camera advanced control functions:

C++

Python

```cpp
#include <dv-processing/io/camera_capture.hpp>

int main() {
    // Open a DVXplorer camera
    dv::io::CameraCapture capture("", dv::io::CameraCapture::CameraType::DVS);

    // Configure event sensitivity to default. Other sensitivities available: VeryLow,
↪ Low, High, VeryHigh
    capture.setDVSBiasSensitivity(dv::io::CameraCapture::BiasSensitivity::Default);

    // Configure event-frame readouts per second (here variable 5000 FPS, the default␣
↪value)
    // See detailed API documentation for other available values
```

(continues on next page)

(continued from previous page)

```
    capture.setDVXplorerEFPS(dv::io::CameraCapture::DVXeFPS::EFPS_VARIABLE_5000);

    // Enable global hold setting (already the default)
    capture.setDVSGlobalHold(true);
    // Disable global reset setting (already the default)
    capture.setDVXplorerGlobalReset(false);

    return 0;
}
```

```
import dv_processing as dv

# Open a DVXplorer camera
capture = dv.io.CameraCapture(dv.io.CameraCapture.CameraType.DVS)

# Configure default event sensitivity. Other sensitivities available: VeryLow, Low, High, VeryHigh
capture.setDVSBiasSensitivity(dv.io.CameraCapture.BiasSensitivity.Default)

# Configure event-frame readouts per second (here variable 5000 FPS, the default value)
# See detailed API documentation for other available values
capture.setDVXplorerEFPS(dv.io.CameraCapture.DVXeFPS.EFPS_VARIABLE_5000)

# Disable global hold setting. Default is True
capture.setDVSGlobalHold(False)
# Enable global reset setting. Default is False
capture.setDVXplorerGlobalReset(True)
```

Read more about DVXplorer biases in our documentation page[9] and specific details about `eFPS` implementation in dv-processing source code[10].

Note: On DVXplorer, setting global hold to false can help for certain applications containing repeating patterns observation, such as flickering LEDs.

DAVIS camera advanced control functions:

- General options:

  C++

  Python

```
#include <dv-processing/io/camera_capture.hpp>

int main() {
    // Open a Davis camera
    dv::io::CameraCapture capture("", dv::io::CameraCapture::CameraType::DAVIS);

    // Setting camera readout to events and frames (default). Other modes available: EventsOnly, FramesOnly
```

(continues on next page)

[9] https://docs.inivation.com/hardware/hardware-advanced-usage/biasing.html#dvxplorer-biases
[10] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.7/include/dv-processing/io/camera_capture.hpp?ref_type=heads#L946-1038

(continued from previous page)

```
    capture.
↪setDavisReadoutMode(dv::io::CameraCapture::DavisReadoutMode::EventsAndFrames);
    // Configure frame output mode to color (default), only on COLOR cameras.␣
↪Other mode available: Grayscale
    capture.setDavisColorMode(dv::io::CameraCapture::DavisColorMode::Color);

    return 0;
}
```

```
import dv_processing as dv

# Open a Davis camera
capture = dv.io.CameraCapture(dv.io.CameraCapture.CameraType.DAVIS)

# Setting camera readout to events and frames (default). Other modes available:␣
↪EventsOnly, FramesOnly
capture.setDavisReadoutMode(dv.io.CameraCapture.DavisReadoutMode.EventsAndFrames)
# Configure frame output mode to color (default), only on COLOR cameras. Other␣
↪mode available: Grayscale
capture.setDavisColorMode(dv.io.CameraCapture.DavisColorMode.Color)
```

- Frame options:

  C++

  Python

```
#include <dv-processing/io/camera_capture.hpp>

#include <chrono>

int main() {
    using namespace std::chrono_literals;

    // Open a Davis camera
    dv::io::CameraCapture capture("", dv::io::CameraCapture::CameraType::DAVIS);

    // Enable frame auto-exposure (default behavior)
    capture.enableDavisAutoExposure();
    // Disable auto-exposure, set frame exposure (here 10ms)
    capture.setDavisExposureDuration(dv::Duration(10ms));
    // Read current frame exposure duration value
    std::optional<dv::Duration> duration = capture.getDavisExposureDuration();
    // Set frame interval duration (here 33ms for ~30FPS)
    capture.setDavisFrameInterval(dv::Duration(33ms));
    // Read current frame interval duration value
    std::optional<dv::Duration> interval = capture.getDavisFrameInterval();

    return 0;
}
```

```
import dv_processing as dv

# Open a Davis camera
capture = dv.io.CameraCapture(dv.io.CameraCapture.CameraType.DAVIS)

# Enable frame auto-exposure (default behavior)
```

(continues on next page)

(continued from previous page)

```
capture.enableDavisAutoExposure()
# Disable auto-exposure, set frame exposure (here 10ms)
capture.setDavisExposureDuration(10000)
# Read current frame exposure duration value
duration = capture.getDavisExposureDuration()
# Set frame interval duration (here 33ms for ~30FPS)
capture.setDavisFrameInterval(33000)
# Read current frame interval duration value
interval = capture.getDavisFrameInterval()
```

- Event options (biases):

> **Warning:** Before using biases, make sure that you absolutely need to change them and that you understand them by reading about biases on our documentation page[11].

C++

Python

```
#include <dv-processing/io/camera_capture.hpp>

int main() {
    // Open a Davis camera
    dv::io::CameraCapture capture("", dv::io::CameraCapture::CameraType::DAVIS);

    /// Access biases raw value
    // Photoreceptor bias
    uint16_t defaultPrBpInt = capture.deviceConfigGet(DAVIS_CONFIG_BIAS, DAVIS346_
↪CONFIG_BIAS_PRBP);
    // Source follower bias
    uint16_t defaultPrSfBpInt = capture.deviceConfigGet(DAVIS_CONFIG_BIAS,␣
↪DAVIS346_CONFIG_BIAS_PRSFBP);
    // Differential bias
    uint16_t defaultDiffBnInt = capture.deviceConfigGet(DAVIS_CONFIG_BIAS,␣
↪DAVIS346_CONFIG_BIAS_DIFFBN);
    // On threshold bias
    uint16_t defaultOnBnInt = capture.deviceConfigGet(DAVIS_CONFIG_BIAS, DAVIS346_
↪CONFIG_BIAS_ONBN);
    // Off threshold bias
    uint16_t defaultOffBnInt = capture.deviceConfigGet(DAVIS_CONFIG_BIAS,␣
↪DAVIS346_CONFIG_BIAS_OFFBN);
    // Refractory period bias
    uint16_t defaultRefrBpInt = capture.deviceConfigGet(DAVIS_CONFIG_BIAS,␣
↪DAVIS346_CONFIG_BIAS_REFRBP);

    /// Change biases values
    // Convert bias integer to values
    caer_bias_coarsefine coarseFinePrBp   =␣
↪caerBiasCoarseFineParse(defaultPrBpInt);
    caer_bias_coarsefine coarseFinePrSfBp =␣
↪caerBiasCoarseFineParse(defaultPrSfBpInt);
    caer_bias_coarsefine coarseFineDiffBn =␣
↪caerBiasCoarseFineParse(defaultDiffBnInt);
```

(continues on next page)

[11] https://docs.inivation.com/hardware/hardware-advanced-usage/biasing.html

(continued from previous page)

```
    caer_bias_coarsefine coarseFineOnBn   = caerBiasCoarseFineParse(defaultOnBnInt);
    caer_bias_coarsefine coarseFineOffBn  = caerBiasCoarseFineParse(defaultOffBnInt);
    caer_bias_coarsefine coarseFineRefrBp = caerBiasCoarseFineParse(defaultRefrBpInt);
    // For example here, add 1 on the log-scale coarse value, i.e. multiply bias value by 10 (approximately)
    coarseFinePrBp.coarseValue   += 1;
    coarseFinePrSfBp.coarseValue += 1;
    coarseFineDiffBn.coarseValue += 1;
    coarseFineOnBn.coarseValue   += 1;
    coarseFineOffBn.coarseValue  += 1;
    coarseFineRefrBp.coarseValue += 1;
    // Convert back
    const uint16_t newPrBp   = caerBiasCoarseFineGenerate(coarseFinePrBp);
    const uint16_t newPrSfBp = caerBiasCoarseFineGenerate(coarseFinePrSfBp);
    const uint16_t newDiffBn = caerBiasCoarseFineGenerate(coarseFineDiffBn);
    const uint16_t newOnBn   = caerBiasCoarseFineGenerate(coarseFineOnBn);
    const uint16_t newOffBn  = caerBiasCoarseFineGenerate(coarseFineOffBn);
    const uint16_t newRefrBp = caerBiasCoarseFineGenerate(coarseFineRefrBp);

    /// Set biases raw value
    // Setting photoreceptor bias
    capture.deviceConfigSet(DAVIS_CONFIG_BIAS, DAVIS346_CONFIG_BIAS_PRBP, newPrBp);
    // Setting source follower bias
    capture.deviceConfigSet(DAVIS_CONFIG_BIAS, DAVIS346_CONFIG_BIAS_PRSFBP, newPrSfBp);
    // Setting differential bias
    capture.deviceConfigSet(DAVIS_CONFIG_BIAS, DAVIS346_CONFIG_BIAS_DIFFBN, newDiffBn);
    // Setting on threshold bias
    capture.deviceConfigSet(DAVIS_CONFIG_BIAS, DAVIS346_CONFIG_BIAS_ONBN, newOnBn);
    // Setting off threshold bias
    capture.deviceConfigSet(DAVIS_CONFIG_BIAS, DAVIS346_CONFIG_BIAS_OFFBN, newOffBn);
    // Setting refractory period bias
    capture.deviceConfigSet(DAVIS_CONFIG_BIAS, DAVIS346_CONFIG_BIAS_REFRBP, newRefrBp);

    return 0;
}
```

Note: Hard-coded bias addresses come from their definition in libcaer as seen here[12].

Note: Conversion utilities `caer_bias_coarse_fine_generate`, `caer_bias_coarse_fine_parse` and `CaerBiasCoarseFine` implementations don't exist in Python since they originally come from libcaer. (struct source code[13], functions source code[14])

[12] https://gitlab.com/inivation/dv/libcaer/-/blob/master/include/libcaer/devices/davis.h#L1334-1372
[13] https://gitlab.com/inivation/dv/libcaer/-/blob/master/include/libcaer/devices/davis.h#L1649-1662
[14] https://gitlab.com/inivation/dv/libcaer/-/blob/master/src/davis.c#L805-840

They are therefore provided as part of this sample.

```python
import dv_processing as dv

# Open a Davis camera
capture = dv.io.CameraCapture(dv.io.CameraCapture.CameraType.DAVIS)

# - Access biases raw value
# Photoreceptor bias
default_pr_bp_int = capture.deviceConfigGet(5, 14)
# Source follower bias
default_pr_sf_bp_int = capture.deviceConfigGet(5, 15)
# Differential bias
default_diff_bn_int = capture.deviceConfigGet(5, 10)
# On threshold bias
default_on_bn_int = capture.deviceConfigGet(5, 11)
# Off threshold bias
default_off_bn_int = capture.deviceConfigGet(5, 12)
# Refractory period bias
default_refr_bp_int = capture.deviceConfigGet(5, 16)


class CaerBiasCoarseFine:
    # Coarse current, from 0 to 7, creates big variations in output current.
    coarse_value: int
    # Fine current, from 0 to 255, creates small variations in output current.
    fine_value: int
    # Whether this bias is enabled or not.
    enabled: bool
    # Bias sex: true for 'N' type, false for 'P' type.
    sex_n: bool
    # Bias type: true for 'Normal', false for 'Cascode'.
    type_normal: bool
    # Bias current level: true for 'Normal, false for 'Low'.
    current_level_normal: bool


def caer_bias_coarse_fine_generate(all_bias_values: CaerBiasCoarseFine) -> int:
    """
    Generate the actual bias integer value to be passed to the device from the different bias components.
    :param all_bias_values: structure containing the different bias value components (see 'CaerBiasCoarseFine' class)
    :return: the bias value as integer to be passed to the device
    """
    bias_value = 0
    # Build up bias value from all its components.
    if all_bias_values.enabled:
        bias_value |= 0x01
    if all_bias_values.sex_n:
        bias_value |= 0x02
    if all_bias_values.type_normal:
        bias_value |= 0x04
    if all_bias_values.current_level_normal:
        bias_value |= 0x08

    bias_value = bias_value | ((all_bias_values.fine_value & 0xFF) << 4)
```

(continues on next page)

(continued from previous page)

```
    bias_value = bias_value | ((all_bias_values.coarse_value & 0x07) << 12)

    return bias_value


def caer_bias_coarse_fine_parse(bias_value: int) -> CaerBiasCoarseFine:
    """
    Extracts the different bias value components from the bias value integer as stored on the device.
    :param bias_value: the bias value as integer as stored on the device
    :return: the structure containing the different bias value components (see 'CaerBiasCoarseFine' class)
    """
    # Decompose bias integer into its parts.
    all_bias_values = CaerBiasCoarseFine()
    all_bias_values.coarse_value = bias_value & 0x01
    all_bias_values.fine_value = bias_value & 0x02
    all_bias_values.enabled = bool(bias_value & 0x04)
    all_bias_values.sex_n = bool(bias_value & 0x08)
    all_bias_values.type_normal = bool((bias_value >> 4) & 0xFF)
    all_bias_values.current_level_normal = bool((bias_value >> 12) & 0x07)
    return all_bias_values


# - Change biases values
# Convert bias integer to values
coarse_fine_pr_bp = caer_bias_coarse_fine_parse(default_pr_bp_int)
coarse_fine_pr_sf_bp = caer_bias_coarse_fine_parse(default_pr_sf_bp_int)
coarse_fine_diff_bn = caer_bias_coarse_fine_parse(default_diff_bn_int)
coarse_fine_on_bn = caer_bias_coarse_fine_parse(default_on_bn_int)
coarse_fine_off_bn = caer_bias_coarse_fine_parse(default_off_bn_int)
coarse_fine_refr_bp = caer_bias_coarse_fine_parse(default_refr_bp_int)
# For example here, add 1 on the log-scale coarse value, i.e. multiply bias value by 10 (approximately)
coarse_fine_pr_bp.coarse_value += 1
coarse_fine_pr_sf_bp.coarse_value += 1
coarse_fine_diff_bn.coarse_value += 1
coarse_fine_on_bn.coarse_value += 1
coarse_fine_off_bn.coarse_value += 1
coarse_fine_refr_bp.coarse_value += 1
# Convert back
new_pr_bp_value = caer_bias_coarse_fine_generate(coarse_fine_pr_bp)
new_pr_sf_bp_value = caer_bias_coarse_fine_generate(coarse_fine_pr_sf_bp)
new_diff_bn = caer_bias_coarse_fine_generate(coarse_fine_diff_bn)
new_on_bn = caer_bias_coarse_fine_generate(coarse_fine_on_bn)
new_off_bn = caer_bias_coarse_fine_generate(coarse_fine_off_bn)
new_refr_bp = caer_bias_coarse_fine_generate(coarse_fine_refr_bp)

# - Set biases raw value
# Setting photoreceptor bias
capture.deviceConfigSet(5, 14, new_pr_bp_value)
# Setting source follower bias
capture.deviceConfigSet(5, 15, new_pr_sf_bp_value)
# Setting differential bias
capture.deviceConfigSet(5, 10, new_diff_bn)
# Setting on threshold bias
capture.deviceConfigSet(5, 11, new_on_bn)
```

(continues on next page)

(continued from previous page)

```
# Setting off threshold bias
capture.deviceConfigSet(5, 12, new_off_bn)
# Setting refractory period bias
capture.deviceConfigSet(5, 16, new_refr_bp)
```

Read events from a live camera

Incoming data from a camera can be read sequentially using the *dv::io::CameraCapture::getNextEventBatch()*. Following is a minimal sample on how to read events sequentially from a camera:

C++

Python

```
#include <dv-processing/io/camera_capture.hpp>

#include <chrono>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Run the loop while camera is still connected
    while (capture.isRunning()) {
        // Read batch of events, check whether received data is correct.
        // The method does not wait for data arrive, it returns immediately with
        // the latest available data or if no data is available, returns a
↪`std::nullopt`.
        if (const auto events = capture.getNextEventBatch(); events.has_value()) {
            // Print received packet information
            std::cout << *events << std::endl;
        }
        else {
            // No data has arrived yet, short sleep to reduce CPU load.
            std::this_thread::sleep_for(1ms);
        }
    }

    return 0;
}
```

```
import dv_processing as dv
import time

# Open any camera
capture = dv.io.CameraCapture()

# Run the loop while camera is still connected
while capture.isRunning():
    # Read batch of events
    events = capture.getNextEventBatch()

    # The method does not wait for data arrive, it returns immediately with
```

(continues on next page)

(continued from previous page)

```
    # latest available data or if no data is available, returns a `None`
    if events is not None:
        # Print received packet time range
        print(f"Received events within time range [{events.getLowestTime()}; {events.
↪getHighestTime()}]")
    else:
        # No data has arrived yet, short sleep to reduce CPU load
        time.sleep(0.001)
```

Read frames from a live camera

Incoming frames from a camera can be read sequentially frame-by-frame using the *dv::io::CameraCapture::getNextFrame()*. Following is a minimal sample on how to read frames sequentially from a camera:

C++

Python

```
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Open any camera
    dv::io::CameraCapture capture;

    // Initiate a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Run the loop while camera is still connected
    while (capture.isRunning()) {
        // Read a frame, check whether it is correct.
        // The method does not wait for frame arrive, it returns immediately with
        // the latest available frame or if no data is available, returns a␣
↪`std::nullopt`.
        if (const auto frame = capture.getNextFrame(); frame.has_value()) {
            std::cout << *frame << std::endl;

            // Show a preview of the image
            cv::imshow("Preview", frame->image);
        }
        cv::waitKey(2);
    }

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv

# Open any camera
capture = dv.io.CameraCapture()

# Initiate a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)
```

(continues on next page)

(continued from previous page)

```python

# Run the loop while camera is still connected
while capture.isRunning():
    # Read a frame from the camera
    frame = capture.getNextFrame()

    # The method does not wait for frame arrive, it returns immediately with
    # latest available frame or if no data is available, returns a `None`
    if frame is not None:
        # Print received packet time range
        print(f"Received a frame at time [{frame.timestamp}]")

        # Show a preview of the image
        cv.imshow("Preview", frame.image)
    cv.waitKey(2)
```

Read IMU data from a live camera

Incoming imu data from a camera can be read sequentially using the *dv::io::CameraCapture::getNextImuBatch()*. Following is a minimal sample on how to read imu data sequentially from a camera:

C++

Python

```cpp
#include <dv-processing/io/camera_capture.hpp>

#include <chrono>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Run the loop while camera is still connected
    while (capture.isRunning()) {
        // Read IMU measurement batch, check whether it is correct.
        // The method does not wait for data to arrive, it returns immediately with
        // the latest available imu data or if no data is available, returns a `std::nullopt`.
        if (const auto imuBatch = capture.getNextImuBatch(); imuBatch.has_value() && !imuBatch->empty()) {
            std::cout << "Received " << imuBatch->size() << " IMU measurements" << std::endl;
        }
        else {
            // No data has arrived yet, short sleep to reduce CPU load.
            std::this_thread::sleep_for(1ms);
        }
    }

    return 0;
}
```

```python
import time
import dv_processing as dv

# Open any camera
capture = dv.io.CameraCapture()

# Run the loop while camera is still connected
while capture.isRunning():
    # Read a batch of IMU data from the camera
    imu_batch = capture.getNextImuBatch()

    # The method does not wait for data to arrive, it returns immediately with
    # latest available data or if no data is available, returns a `None`
    if imu_batch is not None and len(imu_batch) > 0:
        # Print the time range of imu data
        print(f"Received imu data within time range [{imu_batch[0].timestamp}; {imu_batch[-1].timestamp}]")
    else:
        time.sleep(0.001)
```

Read triggers from a live camera

Note: To understand what triggers are and where they come from, read more about them on our documentation page[15].

Incoming trigger data from a camera can be read sequentially using the *dv::io::CameraCapture::getNextTriggerBatch()*. Following is a minimal sample on how to read trigger data sequentially from a camera:

C++

Python

```cpp
#include <dv-processing/io/camera_capture.hpp>

#include <chrono>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Depending on the incoming signal, enable the detection of the desired type of pattern, here we enable everything.
    // Note: In the following variables, replace 'DVX' with 'DAVIS_CONFIG' in case the device used is a DAVIS.
    // Enable rising edge detection
    capture.deviceConfigSet(DVX_EXTINPUT, DVX_EXTINPUT_DETECT_RISING_EDGES, true);
    // Enable falling edge detection
    capture.deviceConfigSet(DVX_EXTINPUT, DVX_EXTINPUT_DETECT_FALLING_EDGES, true);
    // Enable pulse detection
    capture.deviceConfigSet(DVX_EXTINPUT, DVX_EXTINPUT_DETECT_PULSES, true);
    // Enable detector
    capture.deviceConfigSet(DVX_EXTINPUT, DVX_EXTINPUT_RUN_DETECTOR, true);
```

(continues on next page)

[15] https://docs.inivation.com/hardware/hardware-advanced-usage/external-camera-sync.html

(continued from previous page)

```cpp

    // Run the loop while camera is still connected
    while (capture.isRunning()) {
        // Read trigger batch, check whether it is correct.
        // The method does not wait for data to arrive, it returns immediately with
        // the latest available data or if no data is available, returns a `std::nullopt`.
        if (const auto triggers = capture.getNextTriggerBatch(); triggers.has_value() && !triggers->empty()) {
            std::cout << "Received " << triggers->size() << " Triggers" << std::endl;
        }
        else {
            // No data has arrived yet, short sleep to reduce CPU load.
            std::this_thread::sleep_for(1ms);
        }
    }

    return 0;
}
```

```python
import time
import dv_processing as dv

# Open any camera
capture = dv.io.CameraCapture()

# Depending on the incoming signal, enable the detection of the desired type of pattern, here we enable everything.
# Enable rising edge detection
capture.deviceConfigSet(4, 1, True)
# Enable falling edge detection
capture.deviceConfigSet(4, 2, True)
# Enable pulse detection
capture.deviceConfigSet(4, 3, True)
# Enable detector
capture.deviceConfigSet(4, 0, True)

# Run the loop while camera is still connected
while capture.isRunning():
    # Read a batch of triggers from the camera
    triggers = capture.getNextTriggerBatch()

    # The method does not wait for data arrive, it returns immediately with
    # latest available data or if no data is available, returns a `None`
    if triggers is not None and len(triggers) > 0:
        # Print the time range of trigger data
        print(f"Received trigger data within time range [{triggers[0].timestamp};{triggers[-1].timestamp}]")
    else:
        time.sleep(0.001)
```

Note: Hard-coded values come from their corresponding definition (as seen in C++ tab) in libcaer source code:

- for DVXplorer, search for 'DVX_EXTINPUT' and '_DETECT' in here[16].

[16] https://gitlab.com/inivation/dv/libcaer/-/blob/master/include/libcaer/devices/dvxplorer.h

- for Davis, search for 'DAVIS_CONFIG_EXTINPUT' and '_DETECT' in here[17].

Sample application - reading data from a live camera

An application reading multiple types of data from a live camera can be found among the code samples in the source code repository of the dv-processing library:

- Camera capture in C++[18]
- Camera capture in Python[19]

3.6.3 From a file

Data from iniVation cameras are usually recorded using the AEDAT4 file format. The dv-processing library provide tools for reading such files. This section contains explanations and samples on how data can be read from AEDAT4 files. More detailed information on the AEDAT4 file format can be found here[20].

Inspecting AEDAT4 files

AEDAT4 file format supports recording of different data streams into single file, multiple cameras are also supported. The library provides a command-line utility for inspection of AEDAT4 files *dv-filestat*, it provides information on available streams recorded in it.

The utility provides information on the size, timestamp information, duration. More information about the utility can be found *here*.

Opening a file

AEDAT4 files can be opened and read using a *dv::io::MonoCameraRecording* class. This class assumes that the recording was performed using a single camera. Following is a minimal sample code on opening a recording and printing information about it.

A file can be opened by providing its path in the filesystem:

C++

Python

```cpp
#include <dv-processing/io/mono_camera_recording.hpp>

int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Get and print the camera name that data from recorded from
    std::cout << "Opened an AEDAT4 file which contains data from [" << reader.
↪getCameraName() << "] camera"
              << std::endl;

```

(continues on next page)

[17] https://gitlab.com/inivation/dv/libcaer/-/blob/master/include/libcaer/devices/davis.h
[18] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.6/samples/io/camera-capture
[19] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.6/python/samples/camera_capture.py
[20] https://docs.inivation.com/software/software-advanced-usage/file-formats/aedat-4.0.html

(continued from previous page)

```
    return 0;
}
```

```
import dv_processing as dv

# Open a file
reader = dv.io.MonoCameraRecording("path/to/file.aedat4")

# Get and print the camera name that data from recorded from
print(f"Opened an AEDAT4 file which contains data from [{reader.getCameraName()}] camera")
```

Checking available streams

Data recordings might contain various data streams. The *dv::io::MonoCameraRecording* provides easy-to-use methods to inspect what data streams are available. Following sample code shows how to check for existence of various data streams:

C++

Python

```
#include <dv-processing/io/mono_camera_recording.hpp>

int main() {
    // Store the file path
    const std::string pathToFile = "path/to/file.aedat4";

    // Open a file
    dv::io::MonoCameraRecording reader(pathToFile);

    // Print file path and camera name
    std::cout << "Available streams in [" << pathToFile << "]:" << std::endl;

    // Check if event stream is available
    if (reader.isEventStreamAvailable()) {
        // Check the resolution of event stream. Since the getEventResolution() method returns
        // a std::optional, we use *operator to get the value. The method returns std::nullopt
        // only in case the stream is unavailable, which is already checked.
        const cv::Size resolution = *reader.getEventResolution();

        // Print that the stream is present and its resolution
        std::cout << "  * Event stream with resolution " << resolution << std::endl;
    }

    // Check if frame stream is available
    if (reader.isFrameStreamAvailable()) {
        // Check the resolution of frame stream. Since the getFrameResolution() method returns
        // a std::optional, we use *operator to get the value. The method returns std::nullopt
        // only in case the stream is unavailable, which is already checked.
        const cv::Size resolution = *reader.getFrameResolution();
```

(continues on next page)

(continued from previous page)

```
        // Print that the stream is available and its resolution
        std::cout << "  * Frame stream with resolution " << resolution << std::endl;
    }

    // Check if IMU stream is available
    if (reader.isImuStreamAvailable()) {
        // Print that the IMU stream is available
        std::cout << "  * IMU stream" << std::endl;
    }

    // Check if trigger stream is available
    if (reader.isTriggerStreamAvailable()) {
        // Print that the trigger stream is available
        std::cout << "  * Trigger stream " << std::endl;
    }

    return 0;
}
```

```
import dv_processing as dv

# Store the file path
path_to_file = "path/to/file.aedat4"

# Open a file
reader = dv.io.MonoCameraRecording(path_to_file)

# Print file path and camera name
print(f"Checking available streams in [{path_to_file}] for camera name [{reader.
↪getCameraName()}]:")

# Check if event stream is available
if reader.isEventStreamAvailable():
    # Check the resolution of event stream
    resolution = reader.getEventResolution()

    # Print that the stream is present and its resolution
    print(f"  * Event stream with resolution [{resolution.width}x{resolution.height}]
↪")

# Check if frame stream is available
if reader.isFrameStreamAvailable():
    # Check the resolution of frame stream
    resolution = reader.getFrameResolution()

    # Print that the stream is available and its resolution
    print(f"  * Frame stream with resolution [{resolution.width}x{resolution.height}]
↪")

# Check if IMU stream is available
if reader.isImuStreamAvailable():
    # Print that the IMU stream is available
    print("  * IMU stream")

# Check if trigger stream is available
```

(continues on next page)

(continued from previous page)

```python
if reader.isTriggerStreamAvailable():
    # Print that the trigger stream is available
    print("  * Trigger stream")
```

The dv-processing library also supports recording of other types, type agnostic methods are available as templated methods in C++, while python only contains a limited set of named methods (since templating is unavailable in python). Following sample show the use of generic method for checking the availability of certain streams with a name and a type:

```cpp
#include <dv-processing/io/mono_camera_recording.hpp>

int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Store some stream name
    const std::string streamName = "poses";

    // Check if such stream name is available and validate the data type of this stream
    if (reader.isStreamAvailable("streamName") && reader.isStreamOfDataType<dv::Pose>("streamName")) {
        std::cout << "The file contains a stream named [" << streamName << "] and of data type [dv::Pose]" << std::endl;
    }

    return 0;
}
```

Read events from a file

Following sample reads events in batches while the stream has available data to read. While reading from a file, the *dv::io::MonoCameraRecording::getNextEventBatch()* will return data until the end of stream is reached, the *dv::io::MonoCameraRecording::isRunning()* method will return a false boolean when the end is reached.

C++

Python

```cpp
#include <dv-processing/io/mono_camera_recording.hpp>

int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Run the loop while data is available
    while (reader.isRunning()) {
        // Read batch of events, check whether received data is correct.
        if (const auto events = reader.getNextEventBatch(); events.has_value()) {
            // Print received event packet information
            std::cout << *events << std::endl;
        }
    }

    return 0;
}
```

```python
import dv_processing as dv

# Open any camera
reader = dv.io.MonoCameraRecording("path/to/file.aedat4")

# Run the loop while camera is still connected
while reader.isRunning():
    # Read batch of events
    events = reader.getNextEventBatch()
    if events is not None:
        # Print received packet time range
        print(f"{events}")
```

Read frames from a file

Following sample reads frames in batches while the stream has available data to read. While reading from a file, the *dv::io::MonoCameraRecording::getNextFrame()* will return a frame until the end of stream is reached, the *dv::io::MonoCameraRecording::isRunning()* method will return a false boolean when the end is reached.

C++

Python

```cpp
#include <dv-processing/io/mono_camera_recording.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Initiate a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Variable to store the previous frame timestamp for correct playback
    std::optional<int64_t> lastTimestamp = std::nullopt;

    // Run the loop while data is available
    while (reader.isRunning()) {
        // Read a frame, check whether it is correct.
        // The method does not wait for frame arrive, it returns immediately with
        // latest available frame or if no data is available, returns a␣
↪`std::nullopt`.
        if (const auto frame = reader.getNextFrame(); frame.has_value()) {
            // Print information about received frame
            std::cout << *frame << std::endl;

            // Show a preview of the image
            cv::imshow("Preview", frame->image);

            // Calculate the delay between last and current frame, divide by 1000 to␣
↪convert microseconds
            // to milliseconds
            const int delay = lastTimestamp.has_value() ? (frame->timestamp -␣
↪*lastTimestamp) / 1000 : 2;
```

(continues on next page)

(continued from previous page)

```
            // Perform the sleep
            cv::waitKey(delay);

            // Store timestamp for the next frame
            lastTimestamp = frame->timestamp;
        }
    }

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv

# Open a file
reader = dv.io.MonoCameraRecording("path/to/file.aedat4")

# Initiate a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Variable to store the previous frame timestamp for correct playback
lastTimestamp = None

# Run the loop while camera is still connected
while reader.isRunning():
    # Read a frame from the camera
    frame = reader.getNextFrame()

    if frame is not None:
        # Print the timestamp of the received frame
        print(f"Received a frame at time [{frame.timestamp}]")

        # Show a preview of the image
        cv.imshow("Preview", frame.image)

        # Calculate the delay between last and current frame, divide by 1000 to␣
↪convert microseconds
        # to milliseconds
        delay = (2 if lastTimestamp is None else (frame.timestamp - lastTimestamp) /␣
↪1000)

        # Perform the sleep
        cv.waitKey(delay)

        # Store timestamp for the next frame
        lastTimestamp = frame.timestamp
```

Read IMU data from a file

Following sample reads imu data in batches while the stream has available data to read. While reading from a file, the *dv::io::MonoCameraRecording::getNextImuBatch()* will return an IMU measurement batch until the end of stream is reached, the *dv::io::MonoCameraRecording::isRunning()* method will return a false boolean when the end is reached.

C++

Python

```cpp
#include <dv-processing/io/mono_camera_recording.hpp>

int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Run the loop while data is available
    while (reader.isRunning()) {
        // Read IMU measurement batch, check whether it is correct.
        if (const auto imuBatch = reader.getNextImuBatch(); imuBatch.has_value() && !
↪imuBatch->empty()) {
            // Print IMU batch information
            std::cout << "Received " << imuBatch->size() << " IMU measurements" <<␣
↪std::endl;
        }
    }

    return 0;
}
```

```python
import dv_processing as dv

# Open a file
reader = dv.io.MonoCameraRecording("path/to/file.aedat4")

# Run the loop while stream contains data
while reader.isRunning():
    # Read a batch of IMU data from the camera
    imu_batch = reader.getNextImuBatch()
    if imu_batch is not None and len(imu_batch) > 0:
        # Print the info of the imu data
        print(f"Received {len(imu_batch)} IMU measurements")
```

Read triggers from a file

Following sample reads triggers in batches while the stream has available data to read. While reading from a file, the *dv::io::MonoCameraRecording::getNextTriggerBatch()* will return an IMU measurement batch until the end of stream is reached, the *dv::io::MonoCameraRecording::isRunning()* method will return a false boolean when the end is reached.

C++

Python

```cpp
#include <dv-processing/io/mono_camera_recording.hpp>

```

(continues on next page)

(continued from previous page)

```cpp
int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Run the loop while data is available
    while (reader.isRunning()) {
        // Read trigger batch, check whether it is correct.
        if (const auto triggers = reader.getNextTriggerBatch(); triggers.has_value() &
↪& !triggers->empty()) {
            // Print the trigger batch information
            std::cout << "Received " << triggers->size() << " triggers" << std::endl;
        }
    }

    return 0;
}
```

```python
import dv_processing as dv

# Open a file
reader = dv.io.MonoCameraRecording("path/to/file.aedat4")

# Run the loop while camera is still connected
while reader.isRunning():
    # Read a a batch of triggers from the camera
    triggers = reader.getNextTriggerBatch()

    # Check whether batch is valid and contains data
    if triggers is not None and len(triggers) > 0:
        # Print the trigger batch information
        print(f"Received {len(triggers)} triggers")
```

[Advanced] Reading custom data types

The previous samples show how to use named functions to read different data types. C++ API provides templated methods to read any type of data. Below is sample that shows how to read data using the generic templated API:

Note: Since templated methods are only available in C++, the generic writing methods are only available in the C++ API.

```cpp
#include <dv-processing/data/timed_keypoint_base.hpp>
#include <dv-processing/io/mono_camera_recording.hpp>

int main() {
    // Open a file
    dv::io::MonoCameraRecording reader("path/to/file.aedat4");

    // Define and contain a stream name in a variable
    const std::string stream = "keypoints";

    // Check whether a timed-keypoint stream is available
    if (!reader.isStreamAvailable(stream) || !reader.isStreamOfDataType
↪<dv::TimedKeyPointPacket>(stream)) {
```

(continues on next page)

(continued from previous page)

```
        throw dv::exceptions::RuntimeError("Stream named 'keypoints' not found");
    }

    // Run the loop while data is available
    while (reader.isRunning()) {
        // Read timed keypoint batch, check whether it is correct.
        if (const auto keypoints = reader.getNextStreamPacket<dv::TimedKeyPointPacket>
↪(stream); keypoints.has_value()) {
            // Print the number of keypoints read
            std::cout << "Read " << keypoints->elements.size() << " timed keypoints" <
↪< std::endl;
        }
    }

    return 0;
}
```

Sample application - reading data from a recorded AEDAT4 file

An application reading multiple types of data from an AEDAT4 file can be found in the source code repository of the dv-processing library:

- AEDAT4 player in C++[21]
- AEDAT4 to CSV converter in Python[22]

3.7 Writing camera data

The dv-processing library provides a convenient way of recording camera data into AEDAT4 files. These files can record multiple data-types and can be read using existing tools in library. DV GUI software also provides ways to playback these recordings.

3.7.1 Writing a file

Files can be written using the *dv::io::MonoCameraWriter* class. Since the writer supports multiple data type configuration, it needs to know apriori what data streams are needed for writing. They are declared using a *dv::io::MonoCameraWriter::Config* structure and is passed to the constructor of the writer class to correctly initialise the required streams in the file header.

[21] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.6/samples/io/aedat4-reader
[22] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.6/python/samples/aedat4_reader.py

Defining output streams from an existing camera reader handle

As mentioned in the previous paragraph, number of output streams and their data-types needs to be known for the writer. Output streams definition can be generated by an input camera capture. In this case, the writer class will inspect what streams the capture can output and will create output streams for them. Following is an example on how to initialize a writer with a camera capture class:

C++

Python

```cpp
#include <dv-processing/io/camera_capture.hpp>
#include <dv-processing/io/mono_camera_writer.hpp>

int main() {
    // Open any camera
    dv::io::CameraCapture capture;

    // Create the writer instance, writer will inspect the capture capabilities and create output
    // streams for all available data streams from the capture instance.
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", capture);

    // Print which streams were configured
    std::cout << std::boolalpha;
    std::cout << "Is event stream available? " << writer.isEventStreamConfigured() << std::endl;
    std::cout << "Is frame stream available? " << writer.isFrameStreamConfigured() << std::endl;
    std::cout << "Is imu stream available? " << writer.isImuStreamConfigured() << std::endl;
    std::cout << "Is trigger stream available? " << writer.isTriggerStreamConfigured() << std::endl;

    return 0;
}
```

```python
import dv_processing as dv

capture = dv.io.CameraCapture()

writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", capture)

print(f"Is event stream available? {str(writer.isEventStreamConfigured())}")
print(f"Is frame stream available? {str(writer.isFrameStreamConfigured())}")
print(f"Is imu stream available? {str(writer.isImuStreamConfigured())}")
print(f"Is trigger stream available? {str(writer.isTriggerStreamConfigured())}")
```

Defining output streams manually

Output streams definition can be manually defined in the config structure, following is a sample that manually defines event, frame, IMU, and trigger data-type streams and creates a writer instance:

C++

Python

```cpp
#include <dv-processing/io/mono_camera_writer.hpp>

int main() {
    dv::io::MonoCameraWriter::Config config("DVXplorer_sample");

    // Sample VGA resolution, same as the DVXplorer camera
    const cv::Size resolution(640, 480);

    // Add an event stream with a resolution
    config.addEventStream(resolution);

    // Add frame stream with a resolution
    config.addFrameStream(resolution);

    // Add IMU stream
    config.addImuStream();

    // Add trigger stream
    config.addTriggerStream();

    // Create the writer instance with the configuration structure
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);

    // Print which streams were configured
    std::cout << std::boolalpha;
    std::cout << "Is event stream available? " << writer.isEventStreamConfigured() <<␣
↪std::endl;
    std::cout << "Is frame stream available? " << writer.isFrameStreamConfigured() <<␣
↪std::endl;
    std::cout << "Is imu stream available? " << writer.isImuStreamConfigured() <<␣
↪std::endl;
    std::cout << "Is trigger stream available? " << writer.
↪isTriggerStreamConfigured() << std::endl;

    return 0;
}
```

```python
import dv_processing as dv

config = dv.io.MonoCameraWriter.Config("DVXplorer_sample")

# Define VGA resolution for this camera
resolution = (640, 480)

# Add an event stream with a resolution
config.addEventStream(resolution)

# Add frame stream with a resolution
config.addFrameStream(resolution)
```

(continues on next page)

(continued from previous page)

```python

# Add IMU stream
config.addImuStream()

# Add trigger stream
config.addTriggerStream()

# Create the writer instance with the configuration structure
writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", config)

print(f"Is event stream available? {str(writer.isEventStreamConfigured())}")
print(f"Is frame stream available? {str(writer.isFrameStreamConfigured())}")
print(f"Is imu stream available? {str(writer.isImuStreamConfigured())}")
print(f"Is trigger stream available? {str(writer.isTriggerStreamConfigured())}")
```

Defining output streams using predefined templates

The *dv::io::MonoCameraWriter* class provides some templates for configuration to reduce the code lines. Following is a sample how to use a named method that generates a config for writing data:

C++

Python

```cpp
#include <dv-processing/io/mono_camera_writer.hpp>

int main() {
    // Create a DVS config - events, imu, and triggers are going to be enabled
    const auto config = dv::io::MonoCameraWriter::DVSConfig("DVXplorer_sample",␣
↪cv::Size(640, 480));

    // Create the writer instance with the configuration structure
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);

    // Print which streams were configured
    std::cout << std::boolalpha;
    std::cout << "Is event stream available? " << writer.isEventStreamConfigured() <<␣
↪std::endl;
    std::cout << "Is frame stream available? " << writer.isFrameStreamConfigured() <<␣
↪std::endl;
    std::cout << "Is imu stream available? " << writer.isImuStreamConfigured() <<␣
↪std::endl;
    std::cout << "Is trigger stream available? " << writer.
↪isTriggerStreamConfigured() << std::endl;

    return 0;
}
```

```python
import dv_processing as dv

# Create a DVS config - events, imu, and triggers are going to be enabled
config = dv.io.MonoCameraWriter.DVSConfig("DVXplorer_sample", (640, 480))

# Create the writer instance with the configuration structure
writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", config)
```

(continues on next page)

(continued from previous page)

```
8
9  # Print which streams were configured
10 print(f"Is event stream available? {str(writer.isEventStreamConfigured())}")
11 print(f"Is frame stream available? {str(writer.isFrameStreamConfigured())}")
12 print(f"Is imu stream available? {str(writer.isImuStreamConfigured())}")
13 print(f"Is trigger stream available? {str(writer.isTriggerStreamConfigured())}")
```

A list of available output stream configurations available under the *dv::io::MonoCameraWriter* class:

- `EventOnlyConfig` - a single output stream "events"
- `FrameOnlyConfig` - a single output stream "frames"
- `DVSConfig` - three output streams: "events", "imu", and "trigger"
- `DAVISConfig` - four output streams: "events", "frames", "imu", and "trigger"

3.7.2 Writing data to the file

The writer instance can be used to write data with the output streams configured. Following sections provide sample code on how to write individual data types using the writer class.

Writing events

The following sample shows an example on how to write event data using the *dv::io::MonoCameraWriter* class:

C++

Python

```
1  #include <dv-processing/data/generate.hpp>
2  #include <dv-processing/io/mono_camera_writer.hpp>
3
4  int main() {
5      // Sample VGA resolution, same as the DVXplorer camera
6      const cv::Size resolution(640, 480);
7
8      // Event only configuration
9      const auto config = dv::io::MonoCameraWriter::EventOnlyConfig("DVXplorer_sample",␣
   ↪resolution);
10
11     // Create the writer instance, it will only have a single event output stream.
12     dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);
13
14     // Write 100 packet of event data
15     for (int i = 0; i < 100; i++) {
16         // EventStore requires strictly monotonically increasing data, generate
17         // a timestamp from the iteration counter value
18         const int64_t timestamp = i * 10000;
19
20         // Generate sample event batch
21         dv::EventStore events = dv::data::generate::dvLogoAsEvents(timestamp,␣
   ↪resolution);
22
23         // Write the packet using the writer, the data is not going be written at the␣
   ↪exact
```

(continues on next page)

(continued from previous page)

```
        // time of the call to this function, it is only guaranteed to be written after
        // the writer instance is destroyed (destructor has completed)
        writer.writeEvents(events);
    }

    return 0;
}
```

```
import dv_processing as dv

# Sample VGA resolution, same as the DVXplorer camera
resolution = (640, 480)

# Event only configuration
config = dv.io.MonoCameraWriter.EventOnlyConfig("DVXplorer_sample", resolution)

# Create the writer instance, it will only have a single event output stream.
writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", config)

# Write 100 packet of event data
for i in range(100):
    # EventStore requires strictly monotonically increasing data, generate
    # a timestamp from the iteration counter value
    timestamp = i * 1000

    # Empty event store
    events = dv.data.generate.dvLogoAsEvents(timestamp, resolution)

    # Write the packet using the writer, the data is not going be written at the exact
    # time of the call to this function, it is only guaranteed to be written after
    # the writer instance is destroyed (destructor has completed)
    writer.writeEvents(events)
```

Writing frames

The following sample shows an example on how to write image frames using the *dv::io::MonoCameraWriter* class:

C++

Python

```
#include <dv-processing/io/mono_camera_writer.hpp>

int main() {
    // Sample VGA resolution, same as the DVXplorer camera
    const cv::Size resolution(640, 480);

    // Frame only configuration
    const auto config = dv::io::MonoCameraWriter::FrameOnlyConfig("DVXplorer_sample", resolution);

    // Create the writer instance, it will only have a single frame output stream.
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);
```

(continues on next page)

(continued from previous page)

```cpp

    // Write 10 image frames
    for (int i = 0; i < 10; i++) {
        // Initialize a white image
        cv::Mat image(resolution, CV_8UC3, cv::Scalar(255, 255, 255));

        // Generate some monotonically increasing timestamp
        const int64_t timestamp = i * 1000;

        // Encapsulate the image in a frame that has a timestamp, this does not copy the pixel data
        dv::Frame frame(timestamp, image);

        // Write the frame
        writer.writeFrame(frame);
    }

    return 0;
}
```

```python
import dv_processing as dv
import numpy as np

# Frame only configuration
config = dv.io.MonoCameraWriter.FrameOnlyConfig("DVXplorer_sample", (640, 480))

# Create the writer instance, it will only have a single frame output stream
writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", config)

# Write 10 image frames
for i in range(10):
    # Initialize a white image
    image = np.full((480, 640, 3), fill_value=255, dtype=np.uint8)

    # Generate some monotonically increasing timestamp
    timestamp = i * 1000

    # Encapsulate the image in a frame that has a timestamp, this does not copy the pixel data
    frame = dv.Frame(timestamp, image)

    # Write the frame
    writer.writeFrame(frame)
```

Writing IMU data

The following sample shows an example on how to write IMU measurement data using the *dv::io::MonoCameraWriter* class:

C++

Python

```cpp
#include <dv-processing/data/generate.hpp>
#include <dv-processing/io/mono_camera_writer.hpp>
```

(continues on next page)

(continued from previous page)

```cpp

int main() {
    // IMU data only configuration
    auto config = dv::io::MonoCameraWriter::Config("DVXplorer_sample");
    config.addImuStream();

    // Create the writer instance, it will only have a single IMU data output stream
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);

    // Write 100 IMU measurements
    for (int i = 0; i < 100; i++) {
        // Generate some monotonically increasing timestamp
        const int64_t timestamp = i * 1000;

        // Some sample measurements - no rotation on gyro and a one-G gravity on Y␣
↪axis of accelerometer
        const dv::IMU measurement = dv::data::generate::levelImuWithNoise(timestamp);

        // Write the measurement
        writer.writeImu(measurement);
    }

    return 0;
}
```

```python
import dv_processing as dv

# IMU data only configuration
config = dv.io.MonoCameraWriter.Config("DVXplorer_sample")
config.addImuStream()

# Create the writer instance, it will only have a single IMU data output stream
writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", config)

# Write 100 IMU measurements
for i in range(100):
    # Generate some monotonically increasing timestamp
    timestamp = i * 1000

    # Single IMU measurement instance
    measurement = dv.data.generate.levelImuWithNoise(timestamp)

    # Write the measurement
    writer.writeImu(measurement)
```

Writing trigger data

The following sample shows an example on how to write triggers using the *dv::io::MonoCameraWriter* class:

C++

Python

```cpp
#include <dv-processing/io/mono_camera_writer.hpp>

```

(continues on next page)

(continued from previous page)

```cpp
int main() {
    // Trigger data only configuration
    auto config = dv::io::MonoCameraWriter::Config("DVXplorer_sample");
    config.addTriggerStream();

    // Create the writer instance, it will only have a single trigger output stream
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);

    // Write 100 triggers
    for (int i = 0; i < 100; i++) {
        // Generate some monotonically increasing timestamp
        const int64_t timestamp = i * 1000;

        // Single trigger instance, let's say this is some signal from external source
        dv::Trigger trigger(timestamp, dv::TriggerType::EXTERNAL_GENERATOR_RISING_
↪EDGE);

        // Write the trigger value
        writer.writeTrigger(trigger);
    }

    return 0;
}
```

```python
import dv_processing as dv

# Trigger data only configuration
config = dv.io.MonoCameraWriter.Config("DVXplorer_sample")
config.addTriggerStream()

# Create the writer instance, it will only have a single trigger output stream
writer = dv.io.MonoCameraWriter("mono_writer_sample.aedat4", config)

# Write 100 triggers
for i in range(100):
    # Generate some monotonically increasing timestamp
    timestamp = i * 1000

    # Single trigger instance, let's say this is some signal from external source
    trigger = dv.Trigger(timestamp, dv.TriggerType.EXTERNAL_GENERATOR_RISING_EDGE)

    # Write the trigger value
    writer.writeTrigger(trigger)
```

[Advanced] Writing custom data types

The previous samples show how to use named functions for writing different data types into a file. C++ API provides templated methods to write any type of data. Below is sample that shows how to write data using the generic templated API:

Note: Since templated methods are only available in C++, the generic writing methods are only available in the C++ API.

```cpp
#include <dv-processing/io/mono_camera_writer.hpp>

int main() {
    // Create a stream named "keypoints" for a data type of TimedKeyPoint packet
    auto config = dv::io::MonoCameraWriter::Config("DVXplorer_sample");

    // Define and contain a stream name in a variable
    const std::string streamName = "keypoints";

    // Add an output stream with a packet type and a defined stream name
    config.addStream<dv::TimedKeyPointPacket>(streamName);

    // Initialize the writer
    dv::io::MonoCameraWriter writer("mono_writer_sample.aedat4", config);

    // Let's create 10 packets of key-points
    for (int i = 0; i < 10; i++) {
        // Create a packet for writing
        dv::TimedKeyPointPacket packet;

        // Generate some monotonically increasing timestamp
        const int64_t timestamp = i * 1000;

        // 10 Let's generate 10 key-points for the packet
        for (int y = 0; y < 10; y++) {
            // Using emplace_back to directly allocate and insert the keypoint at the end of the vector
            packet.elements.emplace_back(dv::Point2f(5.f, 5.f), 10.f, 3.f, 1.f, 0, -1, timestamp);
        }

        // Write the packet using, requires a stream name that is the same with the
        writer.writePacket(packet, streamName);
    }

    return 0;
}
```

Sample application - recording data from live camera

An application writing multiple types of data from a live camera can be found among the code samples in the source code repository of the dv-processing library:

- Camera writer in C++[23]
- Camera capture in Python[24]

[23] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.6/samples/io/mono-live-writer
[24] https://gitlab.com/inivation/dv/dv-processing/-/blob/rel_1.6/python/samples/writer_mono_dvs.py

3.8 Network streaming

The dv-processing library an easy to integrate solution to stream data over network TCP connections with optional encryption for security or through local sockets. This achieved by using client-server paradigm implemented using two classes: *dv::io::NetworkWriter* and *dv::io::NetworkReader*. The *dv::io::NetworkWriter* acts as a server, supports multiple concurrent clients, and streams a single data type stream to the clients. The *dv::io::NetworkReader* connects to a server and is able to receive the streamed data.

This tutorial will introduce basic usage of these classes to stream event data using a network writer and receive the same data using network reader.

3.8.1 Network streaming server

The following sample implements a network streaming server using *dv::io::NetworkWriter*, that streams periodic event data, that is synthesized in software:

C++

Python

```cpp
#include <dv-processing/data/generate.hpp>
#include <dv-processing/io/network_writer.hpp>

int main() {
    using namespace std::chrono_literals;

    // Define image space resolution dimensions
    const cv::Size resolution(200, 200);

    // Define output event stream with valid resolution
    const dv::io::Stream stream = dv::io::Stream::EventStream(0, "events", "TEST_DATA
↪", resolution);

    // Initiate the server, needs stream definition to initiate
    dv::io::NetworkWriter server("0.0.0.0", 10101, stream);

    // Print the ready state of the server
    std::cout << "Waiting for connections..." << std::endl;

    // Stream interval defines the packet frequency for this sample
    const dv::Duration streamInterval = 10ms;

    // Starting coordinates of the rectangle data that is going to be sent out in␣
↪this sample
    cv::Point2i offset(0, 0);

    // Rectangle size in pixels
    const cv::Point2i rectSize(20, 20);

    // A boolean variable used to define movement direction of the rectangle
    bool direction = true;

    // Run indefinitely
    while (true) {
        // Do not produce output if there are no connected clients
        if (server.getClientCount() > 0) {
```

(continues on next page)

(continued from previous page)

```
            // Generate the rectangle at given offset position
            const dv::EventStore events = dv::data::generate::eventRectangle(dv::now(), offset, offset + rectSize);

            // Increase or decrease the position coordinates depending on the "direction" boolean
            offset = direction ? offset + cv::Point2i(1, 1) : offset - cv::Point2i(1, 1);

            // Check if the rectangle coordinates reaches borders of the image
            if (offset.x == 0 || offset.y == 0 || offset.x + rectSize.x == resolution.width
                || offset.y + rectSize.y == resolution.height) {
                // Reverse the motion direction
                direction = !direction;
            }

            // Send it out to clients
            server.writeEvents(events);
        }

        // Sleep the application for the streaming interval duration
        std::this_thread::sleep_for(streamInterval);
    }

    return EXIT_SUCCESS;
}
```

```
import dv_processing as dv
from datetime import timedelta
import time

# Define image space resolution dimensions
resolution = (200, 200)

# Define output event stream with valid resolution
stream = dv.io.Stream.EventStream(0, "events", "TEST_DATA", resolution)

# Initiate the server, needs stream definition to initiate
server = dv.io.NetworkWriter("0.0.0.0", 10101, stream)

# Print the ready state of the server
print("Waiting for connections...")

# Stream interval defines the packet frequency for this sample
streamInterval = timedelta(milliseconds=10)

# Starting coordinates of the rectangle data that is going to be sent out in this sample
offset = (0, 0)

# Rectangle size in pixels
rectSize = (20, 20)

# A boolean variable used to define movement direction of the rectangle
direction = True
```

(continues on next page)

(continued from previous page)

```

# Run indefinitely
while True:
    # Do not produce output if there are no connected clients
    if server.getClientCount() > 0:
        # Generate the rectangle at given offset position
        events = dv.data.generate.eventRectangle(dv.now(), offset, (offset[0] + rectSize[0], offset[1] + rectSize[1]))

        # Increase or decrease the position coordinates depending on the "direction" boolean
        offset = (offset[0] + 1, offset[1] + 1) if direction else (offset[0] - 1, offset[1] - 1)

        # Check if the rectangle coordinates reaches borders of the image
        if offset[0] == 0 or offset[1] == 0 or offset[0] + rectSize[0] == resolution[0] or offset[1] + rectSize[1] == \
                resolution[1]:
            # Reverse the motion direction
            direction = not direction

        # Send it out to clients
        server.writeEvents(events)

    # Sleep the application for the streaming interval duration
    time.sleep(streamInterval.total_seconds())
```

Compiling and running the sample above will run an event streaming server, which is accessible on port 10101 on all available network devices. The instance will wait for a client to connect, next chapter will introduce a minimal client application to receive and visualize the streamed data.

3.8.2 Network streaming client

The following code sample provide a basic usage of *dv::io::NetworkReader* to receive network streamed data.

C++

Python

```
#include <dv-processing/io/network_reader.hpp>
#include <dv-processing/visualization/event_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Initiate the client connection to the same port and localhost loopback address
    dv::io::NetworkReader client("127.0.0.1", 10101);

    // Validate that this client is connected to an event data stream
    if (!client.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Server does not provide event data!");
    }

    // Initialize the event visualizer with server reported sensor resolution
```

(continues on next page)

(continued from previous page)

```cpp
    dv::visualization::EventVisualizer visualizer(client.getEventResolution().
↪value());

    // Create a preview window to show the visualized events
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Declare an event stream slicer to synchronized event data packets
    dv::EventStreamSlicer slicer;

    // Perform visualization every 10 milliseconds, which should match the server␣
↪publishing frequency
    slicer.doEveryTimeInterval(10ms, [&visualizer](const dv::EventStore &events) {
        // Display preview image
        cv::imshow("Preview", visualizer.generateImage(events));

        // Short sleep, if user clicks escape key (code 27), exit the application
        if (cv::waitKey(2) == 27) {
            exit(0);
        }
    });

    // While client is connected
    while (client.isRunning()) {
        // Read the event data, validate, and feed into the slicer
        if (const auto events = client.getNextEventBatch(); events.has_value()) {
            slicer.accept(*events);
        }
    }

    return EXIT_SUCCESS;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Initiate the client connection to the same port and localhost loopback address
client = dv.io.NetworkReader("127.0.0.1", 10101)

# Validate that this client is connected to an event data stream
if not client.isEventStreamAvailable():
    raise RuntimeError("Server does not provide event data!")

# Initialize the event visualizer with server reported sensor resolution
visualizer = dv.visualization.EventVisualizer(client.getEventResolution())

# Create a preview window to show the visualized events
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Declare an event stream slicer to synchronized event data packets
slicer = dv.EventStreamSlicer()


# Callback method to show the generated event visualization
def show_preview(events: dv.EventStore):
    # Display preview image
```

(continues on next page)

(continued from previous page)

```
    cv.imshow("Preview", visualizer.generateImage(events))

    # Short sleep, if user clicks escape key (code 27), exit the application
    if cv.waitKey(2) == 27:
        exit(0)


# Perform visualization every 10 milliseconds, which should match the server␣
↪publishing frequency
slicer.doEveryTimeInterval(timedelta(milliseconds=10), show_preview)

# While client is connected
while True:
    # Read the event data
    events = client.getNextEventBatch()

    # Validate the data and feed into the slicer
    if events is not None:
        slicer.accept(events)
```

This code sample connects to the port 10101 on local loopback network, receives event data stream and uses *dv::visualization::EventVisualizer* to generate a preview image of the event data.

3.8.3 Running the samples

We can now combine both samples to stream the synthetic events and visualize the output. First, run the server sample application, when it prints "Waiting for connections…", launch the second client application, it should show the preview window with a small rectangle moving from corner to corner.

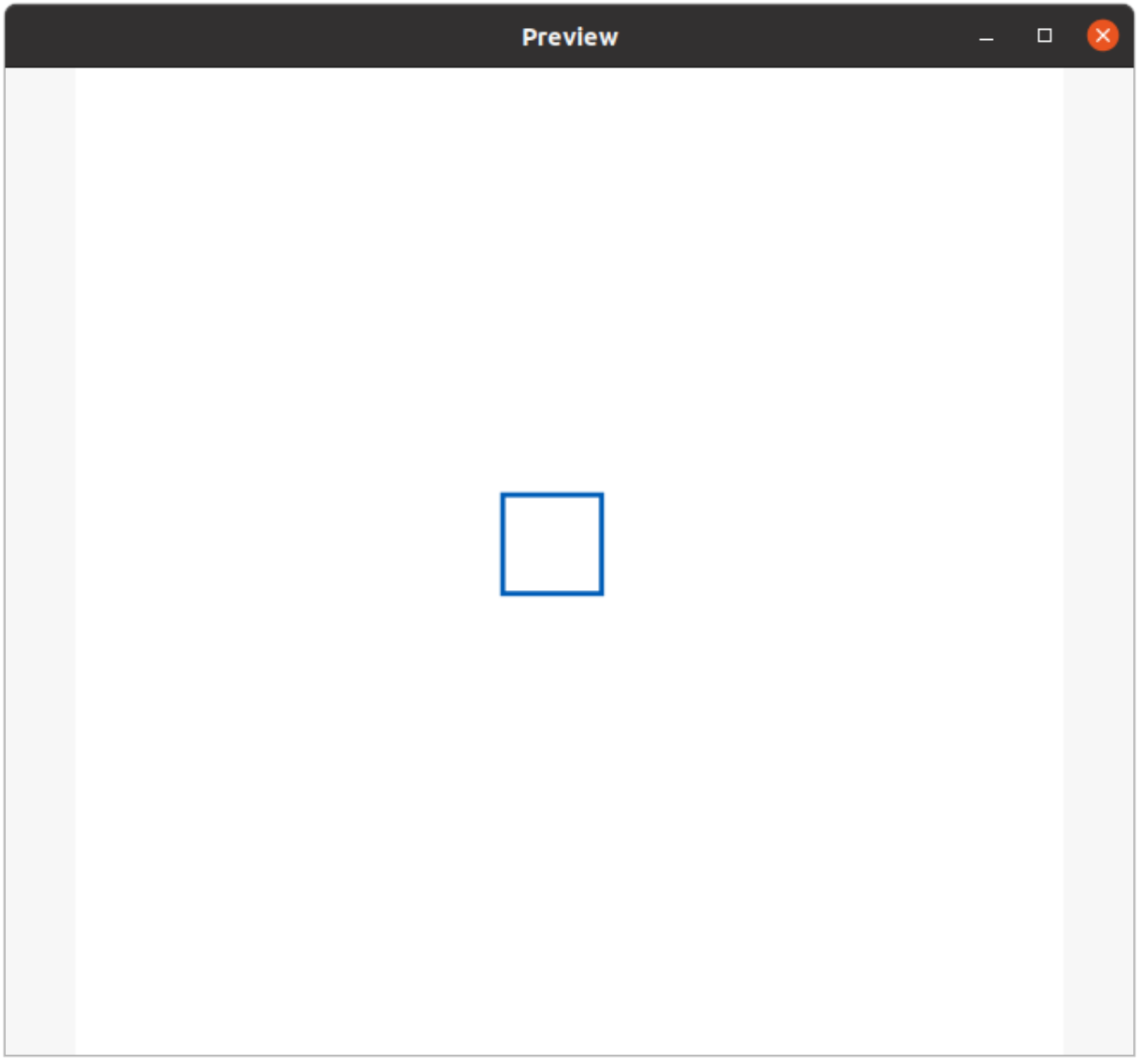


Fig. 17: Preview of the rectangle that is received as event packets, the triangle is expected to move diagonally through the window pixel space.

CHAPTER

FOUR

VISION ALGORITHMS

This chapter describes the available event processing algorithms that can are building blocks for computer vision with event cameras. Most notable algorithms and features:

- Camera geometry - sensor calibration, pixel projections, lens undistortion operations;
- Feature detection and tracking
- Minimal kinematics routines
- Mean-shift clustering

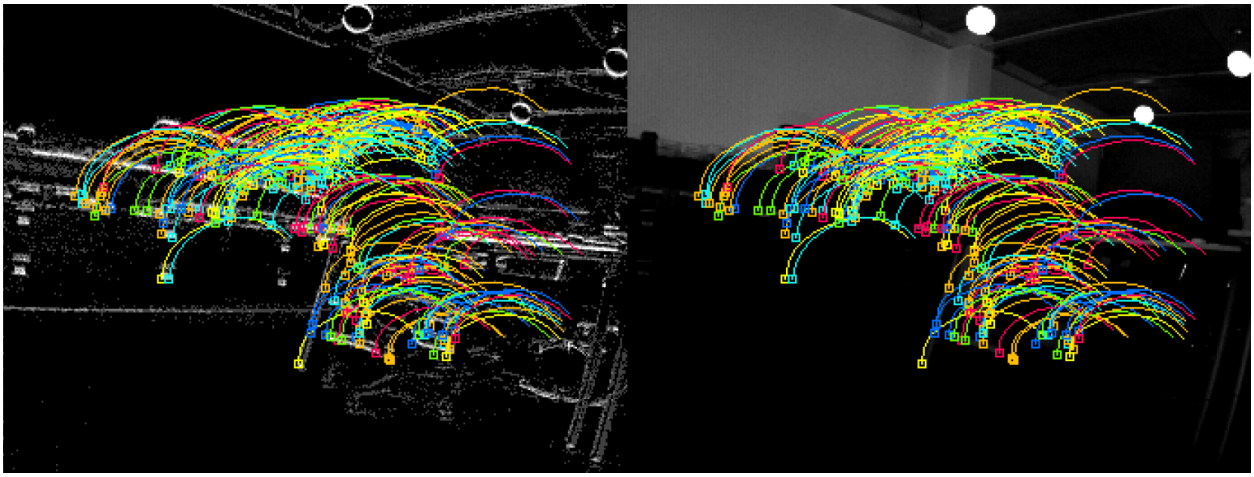


Fig. 1: Tracked features on frame and event streams from a camera.

4.1 Camera Geometry

This section introduces camera calibration related classes and primitives available in the dv-processing library.

4.1.1 Calibration files

Multiple calibration file formats were introduced in different DV software components. Single / stereo camera calibration files were introduced in DV camera calibration module, which produces these files after a successful calibration. A new yaml calibration file format is introduced in dv-processing, which can store calibration information not only for a single / stereo camera setups, but multi-camera setups with any number of cameras as well as calibration for IMU, and it's noise parameters.

All of these calibration files can be opened and created by a provided *dv::camera::CalibrationSet*. The `CalibrationSet` class contains three main types of calibration parameters:

- *dv::camera::calibrations::CameraCalibration* - Single camera intrinsic calibration: pinhole camera and distortion model parameters. It also contains extrinsic transformation matrix which describes physical displacement of a camera w.r.t. one of the cameras. Usually first camera denoted "C0" is selected as a reference which will have an identity matrix for its transformation and all the other cameras will contain transformation matrices which describes the relationship to the first camera.
- *dv::camera::calibrations::StereoCalibration* - Stereo camera extrinsic parameters, this class contains essential and fundamental matrices between a select pair of cameras.
- *dv::camera::calibrations::IMUCalibration* - IMU extrinsic calibration and IMU noise parameters. This class contains extrinsic parameters - transformation between camera plane and timestamp offset (time calibration). It contains IMU measurement noise parameters: gyroscope and accelerometer measurement biases, noise densities, and random walk (white noise) parameters.

The *dv::camera::CalibrationSet* contains sets of these three types of calibration parameters, which is sufficient for storing parameters of any generic visual-inertial multi-camera rig calibration. To see the full list of exact available parameters, please see the list of public members of each of the calibration classes.

Writing a calibration file

The following code sample shows how to generate a calibration file given hardcoded calibration parameters. The generated file is going to be used as input for the next section *Reading calibration file*.

C++

Python

```cpp
#include <dv-processing/camera/calibration_set.hpp>

int main() {
    // Initialize a calibration set
    dv::camera::CalibrationSet calibration;

    // Add a camera calibration with hardcoded calibration parameters, the exact values are just for illustration.
    calibration.addCameraCalibration(dv::camera::calibrations::CameraCalibration("DVXplorer_DXA000312", "left", true,
        cv::Size(640, 480), cv::Point2f(320, 240), cv::Point2f(640, 640), {}, dv::camera::DistortionModel::None,
        std::vector<float>({1.f, 0.f, 0.f, 0.f, 0.f, 1.f, 0.f, 0.f, 0.f, 0.f, 1.f, 0.f, 0.f, 0.f, 0.f, 1.f}),
        dv::camera::calibrations::CameraCalibration::Metadata()));

    // Add an IMU calibration as well, the exact values are just for illustration of use here.
    calibration.addImuCalibration(dv::camera::calibrations::IMUCalibration("DVXplorer_DXA000312", 100.f, 98.1f,
```

(continues on next page)

(continued from previous page)

```
        cv::Point3f(0.f, 0.f, 0.f), cv::Point3f(0.f, 0.f, 0.f), 1.f, 1.f, 1.f, 1.f, 1.
↪f, 1.f, 3500,
        std::vector<float>({1.f, 0.f, 0.f, 0.f, 0.f, 1.f, 0.f, 0.f, 0.f, 0.f, 1.f, 0.
↪f, 0.f, 0.f, 0.f, 1.f}),
        dv::camera::calibrations::IMUCalibration::Metadata()));

    // Just write the generated calibration set into a file.
    calibration.writeToFile("calibration.json");

    return 0;
}
```

```
import dv_processing as dv

calibration = dv.camera.CalibrationSet()

calibration.addCameraCalibration(
    dv.camera.calibrations.CameraCalibration(
        "DVXplorer_DXA000312", "left", True, (640, 480), (320, 240), (640, 640), [],␣
↪dv.camera.DistortionModel.NONE,
        [1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.
↪0],
        dv.camera.calibrations.CameraCalibration.Metadata()))

calibration.addImuCalibration(
    dv.camera.calibrations.IMUCalibration(
        "DVXplorer_DXA000312", 100.0, 98.1, (0.0, 0.0, 0.0), (0.0, 0.0, 0.0), 1.0, 1.
↪0, 1.0, 1.0, 1.0, 1.0, 3500,
        [1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.
↪0],
        dv.camera.calibrations.IMUCalibration.Metadata()))

calibration.writeToFile("calibration.json")
```

Reading calibration file

The following code sample shows the use of the *dv::camera::CalibrationSet* to load an existing calibration file.

Note: This sample requires an existing "calibration.json" file existing in the directory the binary is executed in. The file can be obtained by running previous sample "Writing a calibration file".

C++

Python

```
#include <dv-processing/camera/calibration_set.hpp>

int main() {
    // Initialize a calibration set
    const auto calibrationSet = dv::camera::CalibrationSet::LoadFromFile("calibration.
↪json");

```

(continues on next page)

(continued from previous page)

```
    // Iterate through available camera calibrations. The designation here is an internal camera abbreviation
    // used to refer to a specific sensor in the camera rig.
    for (const auto &[designation, calibration] : calibrationSet.getCameraCalibrations()) {
        // Print the designation and the camera name of current calibration
        std::cout << "[" << designation << "] Found calibration for camera with name [" << calibration.name << "]"
                  << std::endl;

        // Print the intrinsic calibration parameters for this camera: focal length, principal point, distortion model
        // and parameters of the distortion model
        std::cout << "\t Focal length: " << calibration.focalLength << std::endl;
        std::cout << "\t Principal point: " << calibration.principalPoint << std::endl;
        std::cout << "\t Distortion model: " << calibration.getDistortionModelString() << std::endl;
        std::cout << "\t Distortion parameters: "
                  << fmt::format("[{}]", fmt::join(calibration.distortion.begin(), calibration.distortion.end(), ", "))
                  << std::endl;
    }

    // Iterate through available IMU calibrations in the file
    for (const auto &[designation, calibration] : calibrationSet.getImuCalibrations()) {
        // Print the designation and the camera name of current calibration
        std::cout << "[" << designation << "] Found IMU calibration for camera with name [" << calibration.name << "]"
                  << std::endl;

        // Print some available information: accelerometer and gyroscope measurement limits, calibrated time offset and
        // biases
        std::cout << "\t Maximum acceleration: " << calibration.accMax << " [m/s^2]" << std::endl;
        std::cout << "\t Maximum angular velocity: " << calibration.omegaMax << " [rad/s]" << std::endl;
        std::cout << "\t Time offset: " << calibration.timeOffsetMicros << " [µs]" << std::endl;
        std::cout << "\t Accelerometer bias: " << calibration.accOffsetAvg << " [m/s^2]" << std::endl;
        std::cout << "\t Gyroscope bias: " << calibration.omegaOffsetAvg << " [rad/s]" << std::endl;

        // Print noise density values for the IMU sensor
        std::cout << "\t Accelerometer noise density: " << calibration.accNoiseDensity << " [m/s^2/sqrt(Hz)]"
                  << std::endl;
        std::cout << "\t Gyroscope noise density: " << calibration.omegaNoiseDensity << " [rad/s/sqrt(Hz)]"
                  << std::endl;
    }

    return 0;
}
```

```python
import dv_processing as dv

# Initialize a calibration set
calibration_set = dv.camera.CalibrationSet.LoadFromFile("calibration.json")

# Iterate through available camera calibrations. The designation here is an internal camera abbreviation
# used to refer to a specific sensor in the camera rig.
for designation, calibration in calibration_set.getCameraCalibrations().items():
    # Print the designation and the camera name of current calibration
    print(f"[{designation}] Found calibration for camera with name [{calibration.name}]")

    # Print the intrinsic calibration parameters for this camera: focal length, principal point, distortion model
    # and parameters of the distortion model
    print(f"\t Focal length: {calibration.focalLength}")
    print(f"\t Principal point: {calibration.principalPoint}")
    print(f"\t Distortion model: {calibration.distortionModel}")
    print(f"\t Distortion parameters: {calibration.distortion}")

# Iterate through available IMU calibrations in the file
for designation, calibration in calibration_set.getImuCalibrations().items():
    # Print the designation and the camera name of current calibration
    print(f"[{designation}] Found IMU calibration for camera with name [{calibration.name}]")

    # Print some available information: accelerometer and gyroscope measurement limits, calibrated time offset and
    # biases
    print(f"\t Maximum acceleration: {calibration.accMax} [m/s^2]")
    print(f"\t Maximum angular velocity: {calibration.omegaMax} [rad/s]")
    print(f"\t Time offset: {calibration.timeOffsetMicros} [µs]")
    print(f"\t Accelerometer bias: {calibration.accOffsetAvg} [m/s^2]")
    print(f"\t Gyroscope bias: {calibration.omegaOffsetAvg} [rad/s]")

    # Print noise density values for the IMU sensor
    print(f"\t Accelerometer noise density: {calibration.accNoiseDensity} [m/s^2/sqrt(Hz)]")
    print(f"\t Gyroscope noise density: {calibration.omegaNoiseDensity} [rad/s/sqrt(Hz)]")
```

4.1.2 Monocular camera geometry

While *dv::camera::CalibrationSet* class is designed for reading / writing the calibration parameters of multi-camera rig and *dv::camera::calibrations::CameraCalibration* stores intrinsic calibration parameters of a single pinhole camera model, these classes do not provide any operations that perform projections and (un)distortion of point coordinates. *dv::camera::CameraGeometry* class provides an efficient implementation of the mathematical operations for:

- Forward / backward projection,
- Distortion model distortion and undistortion.

Note: The operations are intended for sparse point operations, since event-camera produces parse pixel data. For

operations on full image frame prefer the use of highly optimized operations available in OpenCV.

The following code sample shows the use of *dv::camera::CameraGeometry* to back-project pixel coordinates onto 3D space, apply a 3D transformation on the points using the kinematics library and project the points back to pixel coordinates.

C++

Python

```cpp
#include <dv-processing/camera/camera_geometry.hpp>
#include <dv-processing/core/event.hpp>
#include <dv-processing/data/generate.hpp>
#include <dv-processing/kinematics/transformation.hpp>
#include <dv-processing/visualization/colors.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Use VGA resolution for this sample
    const cv::Size resolution(640, 480);

    // Initialize an ideal pinhole camera model parameters with no distortion
    dv::camera::CameraGeometry geometry(640.f, 640.f, 320.f, 240.f, resolution);

    dv::EventStore positiveEvents;

    // Generate a sample set of events and filter out only positive events for this sample
    dv::polarityFilter(dv::data::generate::dvLogoAsEvents(0, resolution), positiveEvents, true);

    // Back project the events into a set of 3D points
    const auto points = geometry.backProjectSequence<std::vector<dv::Point3f>>(positiveEvents);

    // Apply some 3D transformation
    dv::kinematics::Transformationf shift(
        0, Eigen::Vector3f(0.5f, 0.3f, 0.1f), Eigen::Quaternionf(0.24f, -0.31f, -0.89f, 0.18f));

    // Apply the transformation above to each of the back-projected points
    std::vector<dv::Point3f> shiftedPoints;
    for (const auto &point : points) {
        shiftedPoints.push_back(shift.transformPoint<dv::Point3f>(point));
    }

    // Forward project the points with transformation
    const auto rotatedPixels = geometry.projectSequence<std::vector<cv::Point2f>>(shiftedPoints);

    // Choose a color for visualization and store it in a variable that can be efficiently assigned to a pixel intensity
    const auto blue = dv::visualization::colors::iniBlue;
    const cv::Vec3b color(static_cast<uint8_t>(blue[0]), static_cast<uint8_t>(blue[1]), static_cast<uint8_t>(blue[2]));

    // Draw input events on an image for input preview
```

(continues on next page)

(continued from previous page)

```cpp
    cv::Mat input(resolution, CV_8UC3, dv::visualization::colors::white);
    for (const auto &event : positiveEvents) {
        input.at<cv::Vec3b>(event.y(), event.x()) = color;
    }

    // Draw output pixels on another image
    cv::Mat output(resolution, CV_8UC3, dv::visualization::colors::white);
    for (const auto &pixel : rotatedPixels) {
        output.at<cv::Vec3b>(pixel) = color;
    }

    // Concatenate both image for a single image preview
    cv::Mat preview;
    cv::hconcat(input, output, preview);

    // Create preview window and show the image
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);
    cv::imshow("Preview", preview);
    cv::waitKey();

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
import numpy as np

# Use VGA resolution for this sample
resolution = (640, 480)

# Initialize an ideal pinhole camera model parameters with no distortion
geometry = dv.camera.CameraGeometry(640, 640, 320, 240, resolution)

# Generate a sample set of events and filter out only positive events for this sample
positiveEvents = dv.polarityFilter(dv.data.generate.dvLogoAsEvents(0, resolution),␣
↪True)

# Back project the events into a set of 3D points
points = geometry.backProjectSequence(positiveEvents)

# Apply some 3D transformation
shift = dv.kinematics.Transformationf(0, (0.5, 0.3, 0.1), (0.24, -0.31, -0.89, 0.18))

# Apply the transformation above to each of the back-projected points
shiftedPoints = []
for point in points:
    shiftedPoints.append(shift.transformPoint(point))

# Forward project the points with transformation
rotatedPixels = geometry.projectSequence(shiftedPoints)

# Draw input events on an image for input preview
input = np.ndarray((480, 640, 3), dtype=np.uint8)
input.fill(255)
for event in positiveEvents:
    input[event.y(), event.x(), :] = dv.visualization.colors.iniBlue()
```

(continues on next page)

(continued from previous page)

```
33
34 # Draw output pixels on another image
35 output = np.ndarray((480, 640, 3), dtype=np.uint8)
36 output.fill(255)
37 for pixel in rotatedPixels:
38     output[int(pixel[1]), int(pixel[0]), :] = dv.visualization.colors.iniBlue()
39
40 # Create a window and show a concatenated preview image of input events and output␣
   ↪coordinates
41 cv.namedWindow("Preview", cv.WINDOW_NORMAL)
42 cv.imshow("Preview", cv.hconcat([input, output]))
43 cv.waitKey()
```

Fig. 2: Expected result of running the sample code for camera geometry transformation.

4.1.3 Stereo camera geometry

The *dv::camera::StereoGeometry* class is intended for efficient rectification of sparse point coordinates and 3D depth estimation. The *dv::camera::StereoGeometry* is built on top of monocular `dv::io::CameraGeometry` projections and adds sparse stereo rectification that aligns coordinates for stereo disparity matching. The class also provides convenience methods to estimate 3D depth with known disparity.

For sample use of the stereo geometry class, please refer to full disparity estimation samples available here[25].

4.2 Feature detection in events

The dv-processing library provides extensible interfaces and implementations for feature detection on event streams. Features are certain locations of interest in the data that can be detected with repeatability, such as corners. The detected features can be tracked over time in the event stream, so the library provides a reusable interface for feature detection algorithm implementations that can be used with a tracking algorithm.

[25] https://gitlab.com/inivation/dv/dv-processing/-/tree/rel_1.6/samples/depth

4.2.1 Feature detectors

This section describes available feature detectors and their basic usage.

Feature detector using OpenCV

The most basic use of a feature detector is to use OpenCV feature detector implementation with the dv-processing feature detector wrapper on a *dv::Frame*. The feature detection wrapper base class *dv::features::FeatureDetector* was designed to wrap a feature detection algorithm and provide a few additional processing steps to handle image margins (ignore feature that are very close to edge of the image) and subsampling as a post-detection step (sampling only an optimal set of features).

Margins are set by a floating-point coefficient value, which expresses a relative margin size from the resolution of an input image, e.g. 0.05 would set the margins of 5% of the total width / height of the image.

Only a select number of features are useful for most applications. The *dv::features::FeatureDetector* simplifies feature subsampling by providing a post-processing step that sub-samples the output features. The sub-sampling can be performed by enabling one the following post-processing options:

- `None`: Do not perform any post-processing, all features from detection will be returned.
- `TopN`: Retrieve a given number of the highest scoring features.
- `AdaptiveNMS`: Apply the AdaptiveNMS algorithm to retrieve equally spaced features in pixel space dimensions. More information on the AdaptiveNMS here: original code[26] and paper[27] .

The following code sample shows how to detect the "good features to track" from OpenCV on an accumulated image from a live camera:

C++

Python

```cpp
#include <dv-processing/core/frame.hpp>
#include <dv-processing/features/feature_detector.hpp>
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/features2d.hpp>
#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event␣
↪stream.");
    }

    // Initialize an accumulator with camera resolution
    dv::Accumulator accumulator(*capture.getEventResolution());

```

(continues on next page)

[26] https://github.com/BAILOOL/ANMS-Codes

[27] https://www.researchgate.net/publication/323388062_Efficient_adaptive_non-maximal_suppression_algorithms_for_homogeneous_spatial_keypoint_distribution

(continued from previous page)

```cpp
    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Let's detect 100 features
    const size_t numberOfFeatures = 100;

    // Create an image feature detector with given resolution. By default, it uses FAST feature detector
    // with AdaptiveNMS post-processing.
    auto detector = dv::features::ImageFeatureDetector(*capture.getEventResolution(), cv::GFTTDetector::create());

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&accumulator, &detector](const dv::EventStore &events) {
        // Pass events into the accumulator and generate a preview frame
        accumulator.accept(events);
        dv::Frame frame = accumulator.generateFrame();

        // Run the feature detection on the accumulated frame
        const auto features = detector.runDetection(frame, numberOfFeatures);

        // Create a colored preview image by converting from grayscale to BGR
        cv::Mat preview;
        cv::cvtColor(frame.image, preview, cv::COLOR_GRAY2BGR);

        // Draw detected features
        cv::drawKeypoints(preview, dv::data::fromTimedKeyPoints(features), preview);

        // Show the accumulated image
        cv::imshow("Preview", preview);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

```

(continues on next page)

(continued from previous page)

```
# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize an accumulator with some resolution
accumulator = dv.Accumulator(capture.getEventResolution())

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Let's detect 100 features
number_of_features = 100

# Create an image feature detector with given resolution. By default, it uses FAST feature detector
# with AdaptiveNMS post-processing.
detector = dv.features.ImageFeatureDetector(capture.getEventResolution())

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Pass events into the accumulator and generate a preview frame
    accumulator.accept(events)
    frame = accumulator.generateFrame()

    # Run the feature detection on the accumulated frame
    features = detector.runDetection(frame, number_of_features)

    # Create a colored preview image by converting from grayscale to BGR
    preview = cv.cvtColor(frame.image, cv.COLOR_GRAY2BGR)
    for feature in features:
        # Draw a rectangle marker on each feature location
        cv.drawMarker(preview, (int(feature.pt[0]), int(feature.pt[1])),
                      dv.visualization.colors.someNeonColor(feature.class_id), cv.MARKER_SQUARE, 10, 2)

    # Show the accumulated image
    cv.imshow("Preview", preview)
    cv.waitKey(2)


# Register a callback every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

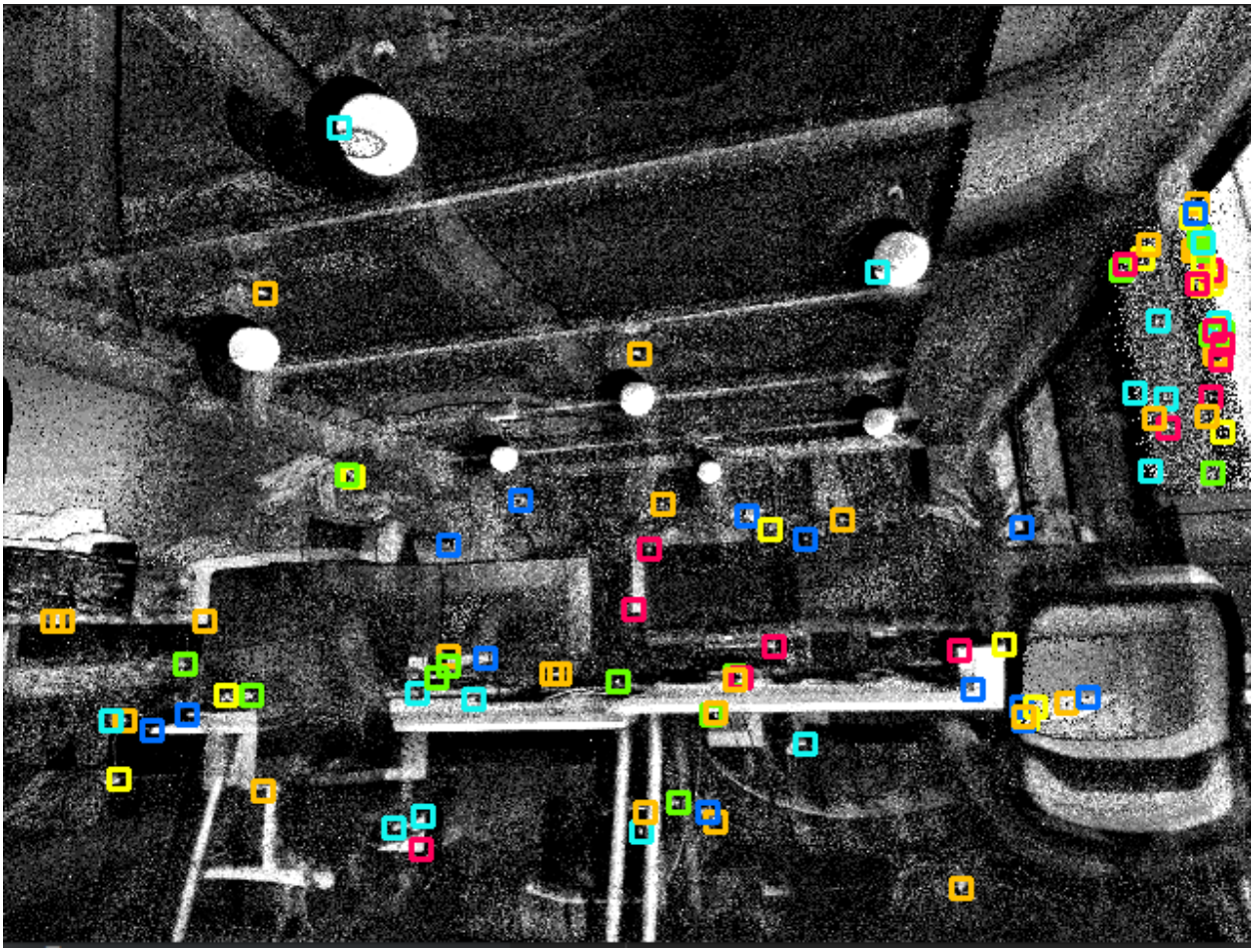


Fig. 3: Detected “good features to track” on an accumulated image.

Arc* Feature detector

The library provides an implementation of Arc* feature detector. Arc* performs corner detection on a per-event basis, so it uses *dv::EventStore* as an input. More details on this feature detection algorithm can be found in the original paper publication[28] .

C++

Python

```cpp
#include <dv-processing/core/frame.hpp>
#include <dv-processing/features/arc_corner_detector.hpp>
#include <dv-processing/features/feature_detector.hpp>
#include <dv-processing/io/camera_capture.hpp>
#include <dv-processing/visualization/colors.hpp>

#include <opencv2/features2d.hpp>
#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    const auto resolution = *capture.getEventResolution();

    // Initialize an accumulator with camera resolution
    dv::Accumulator accumulator(resolution);

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Let's detect 100 features
    const size_t numberOfFeatures = 100;

    // Create an Arc* feature detector with given resolution. Corner range is set to 5 millisecond.
    auto detector = dv::features::FeatureDetector<dv::EventStore, dv::features::ArcCornerDetector<>>(
        resolution, std::make_shared<dv::features::ArcCornerDetector<>>(resolution, 5000, false));

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&accumulator, &detector](const dv::EventStore &events) {
        // Run the feature detection on the incoming events. Features are extracted on a per-event basis
```

(continues on next page)

[28] https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/277131/RAL2018-camera-ready.pdf

(continued from previous page)

```cpp
        const auto features = detector.runDetection(events, numberOfFeatures);

        // Run accumulator for an image, it is going to be used only for preview
        accumulator.accept(events);
        const dv::Frame frame = accumulator.generateFrame();

        // Create a colored preview image by converting from grayscale to BGR
        cv::Mat preview;
        cv::cvtColor(frame.image, preview, cv::COLOR_GRAY2BGR);

        // Draw detected features
        cv::drawKeypoints(preview, dv::data::fromTimedKeyPoints(features), preview);

        // Show the accumulated image
        cv::imshow("Preview", preview);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize an accumulator with some resolution
accumulator = dv.Accumulator(capture.getEventResolution())

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Let's detect 100 features
number_of_features = 100

# Create an Arc* feature detector with given resolution. Corner range is set to 5␣
↪millisecond.
detector = dv.features.ArcEventFeatureDetector(capture.getEventResolution(), 5000,␣
↪False)

# Initialize a slicer
```

(continues on next page)

(continued from previous page)

```
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Run the feature detection on the incoming events
    features = detector.runDetection(events, number_of_features)

    # Pass events into the accumulator and generate a preview frame
    accumulator.accept(events)
    frame = accumulator.generateFrame()

    # Create a colored preview image by converting from grayscale to BGR
    preview = cv.cvtColor(frame.image, cv.COLOR_GRAY2BGR)
    for feature in features:
        # Draw a rectangle marker on each feature location
        cv.drawMarker(preview, (int(feature.pt[0]), int(feature.pt[1])),
                      dv.visualization.colors.someNeonColor(feature.class_id), cv.
↪MARKER_SQUARE, 10, 2)

    # Show the accumulated image
    cv.imshow("Preview", preview)
    cv.waitKey(2)


# Register a callback every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

Note: The provided implementation is not suitable for running real-time with high event rates.

Event-based blob detector

The library provides an implementation of a simple event-based blob detector. The class makes use of `cv::SimpleBlobDetector` to detect blobs on an image. The image is generated using *dv::EdgeMapAccumulator*. A default `cv::SimpleBlobDetector` is provided with reasonable values to safely detect blobs and not noise in the accumulated image. It is also possible to specify the detector that should be used to find the blobs and some specific region of interests in the image plane where the blobs should be searched. In addition to this, eventually, it is also possible to specify the down sampling factor to be applied to the image before performing the detection and also specify any additional pre-processing step that should be applied before running the actual detection step. In summary, the detection steps are as following:

1. Compute accumulated image from events
2. Apply ROI and mask to the accumulated event image

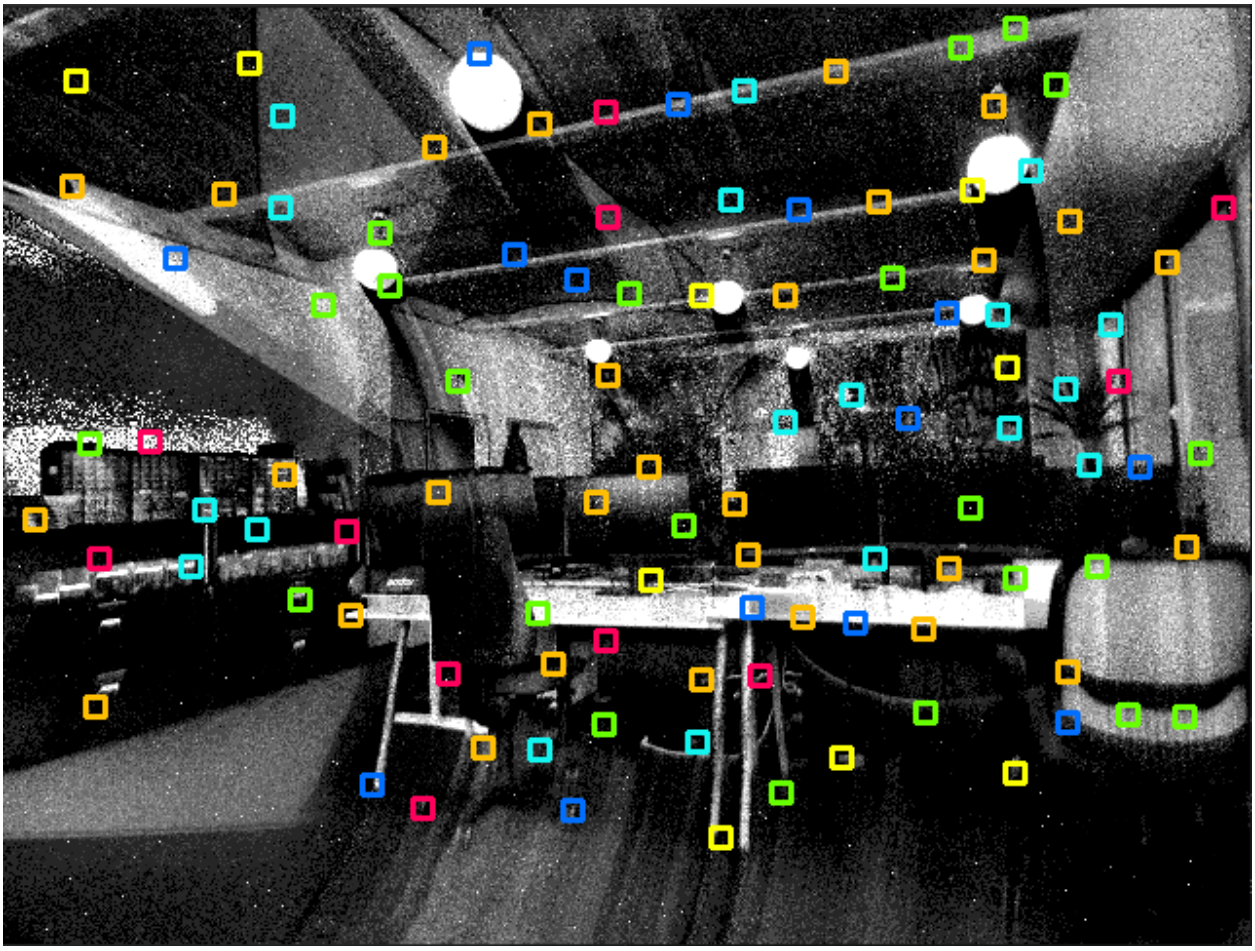


Fig. 4: Detected Arc* features displayed on an accumulated image.

3. Down sample image (optional)
4. Apply pre-process function (optional)
5. Detect blobs
6. Rescale blobs to original resolution (if down sampling was performed)

C++

Python

```cpp
#include <dv-processing/features/event_blob_detector.hpp>
#include <dv-processing/features/feature_detector.hpp>
#include <dv-processing/io/camera_capture.hpp>
#include <dv-processing/visualization/colors.hpp>

#include <opencv2/features2d.hpp>
#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    const auto resolution = *capture.getEventResolution();

    // Initialize an accumulator with camera resolution (for visualization only)
    dv::Accumulator accumulator(resolution);

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Let's detect up to 100 features
    const size_t numberOfFeatures = 100;

    // define preprocessing step (here we do nothing but show how it can be implemented)
    std::function<void(cv::Mat &)> preprocess = [](cv::Mat &image) {
    };

    // Define number of pyr-down to be applied to the image before detecting blobs. If pyramidLevel is set to zero, the
    // blobs will be detected on the original image resolution, if pyramid level is one, the image will be down sampled
    // by a factor of 2 (factor of 4 if pyramid level is 2 and so on...) before performing the detection. The blobs will
    // then be scaled back to the original resolution size.
    const int32_t pyramidLevel = 0;
    // Create an event-based blob detector
    const auto blobDetector = std::make_shared<dv::features::EventBlobDetector>(resolution, pyramidLevel, preprocess);
    auto detector           = dv::features::EventFeatureBlobDetector(resolution,
```

(continues on next page)

(continued from previous page)

```
↪blobDetector);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Register a callback every 33 milliseconds
    slicer.doEveryTimeInterval(33ms, [&accumulator, &detector](const dv::EventStore &
↪events) {
        // Run the feature detection on the incoming events.
        const auto features = detector.runDetection(events, numberOfFeatures);

        // Run accumulator for an image, it is going to be used only for preview
        accumulator.accept(events);
        const dv::Frame frame = accumulator.generateFrame();

        // Create a colored preview image by converting from grayscale to BGR
        cv::Mat preview;
        cv::cvtColor(frame.image, preview, cv::COLOR_GRAY2BGR);

        // Draw detected features
        cv::drawKeypoints(preview, dv::data::fromTimedKeyPoints(features), preview);

        // Show the accumulated image
        cv::imshow("Preview", preview);
        cv::waitKey(2);
    });

    // Run the event processing while the camera is connected
    while (capture.isRunning()) {
        // Receive events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // If so, pass the events into the slicer to handle them
            slicer.accept(*events);
        }
    }

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta


def preprocess(image):
    # add your pre precessing here..
    return image


# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

```

(continues on next page)

(continued from previous page)

```
# Initialize an accumulator with some resolution (for visualization only)
accumulator = dv.Accumulator(capture.getEventResolution())

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Let's detect 100 features
number_of_features = 100

# Define number of pyr-down to be applied to the image before detecting blobs. If pyramidLevel is set to zero, the
# blobs will be detected on the original image resolution, if pyramid level is one, the image will be down sampled by a
# factor of 2 (factor of 4 if pyramid level is 2 and so on..) before perfirming the detection. The blobs will then be
# scaled back to the original resolution size.
pyramidLevel = 0

# Create an event-based blob detector
detector = dv.features.EventFeatureBlobDetector(capture.getEventResolution(), pyramidLevel, preprocess)

# Initialize a slicer
slicer = dv.EventStreamSlicer()


# Declare the callback method for slicer
def slicing_callback(events: dv.EventStore):
    # Run the feature detection on the incoming events
    features = detector.runDetection(events, number_of_features)

    # Pass events into the accumulator and generate a preview frame
    accumulator.accept(events)
    frame = accumulator.generateFrame()

    # Create a colored preview image by converting from grayscale to BGR
    preview = cv.cvtColor(frame.image, cv.COLOR_GRAY2BGR)
    for feature in features:
        # Draw a rectangle marker on each feature location
        cv.drawMarker(preview, (int(feature.pt[0]), int(feature.pt[1])),
                      dv.visualization.colors.someNeonColor(feature.class_id), cv.MARKER_SQUARE, 10, 2)

    # Show the accumulated image
    cv.imshow("Preview", preview)
    cv.waitKey(2)


# Register a callback every 33 milliseconds
slicer.doEveryTimeInterval(timedelta(milliseconds=33), slicing_callback)

# Run the event processing while the camera is connected
while capture.isRunning():
    # Receive events
    events = capture.getNextEventBatch()

    # Check if anything was received
```

(continues on next page)

(continued from previous page)

```
    if events is not None:
        # If so, pass the events into the slicer to handle them
        slicer.accept(events)
```

4.3 Feature tracking

The dv-processing library provides a few algorithm implementations to perform visual tracking of detected features. Feature tracking was intended for use in the frontends of visual odometry pipelines. While tracking on event input is feasible, the library also provides frame-based and hybrid (which uses both events and frames) trackers that allow to build visual odometry pipelines that leverage both input modalities.

4.3.1 Frame-based tracking

Frame based feature tracking is performed by using Lucas-Kanade tracking algorithm. The following sample shows how to use the available frame based tracker with a stream of incoming frames.

The following code sample shows how to run a feature tracker on frames coming from a live camera.

Note: This sample requires a camera that is capable of producing frames, e.g. a DAVIS series camera.

C++

Python

```
#include <dv-processing/features/feature_tracks.hpp>
#include <dv-processing/features/image_feature_lk_tracker.hpp>
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isFrameStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide a frame␣
↪stream.");
    }

    const cv::Size resolution = capture.getFrameResolution().value();

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Instantiate a visual tracker with known resolution, all parameters kept default
    auto tracker = dv::features::ImageFeatureLKTracker::RegularTracker(resolution);

    // Create a track container instance that is used to visualize tracks on an image
    dv::features::FeatureTracks tracks;

```

(continues on next page)

(continued from previous page)

```cpp
    // Run the frame processing while the camera is connected
    while (capture.isRunning()) {
        // Try to receive a frame, check if anything was received
        if (const auto frame = capture.getNextFrame()) {
            // Pass the frame to the tracker
            tracker->accept(*frame);

            // Run tracking
            const auto result = tracker->runTracking();

            // Pass tracking result into the track container which aggregates track history
            tracks.accept(result);

            // Generate and show a preview of recent tracking history
            cv::imshow("Preview", tracks.visualize(frame->image));
        }
        cv::waitKey(2);
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isFrameStreamAvailable():
    raise RuntimeError("Input camera does not provide a frame stream.")

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Instantiate a visual tracker with known resolution, all parameters kept default
tracker = dv.features.ImageFeatureLKTracker.RegularTracker(capture.getEventResolution())

# Create a track container instance that is used to visualize tracks on an image
tracks = dv.features.FeatureTracks()

# Run the frame processing while the camera is connected
while capture.isRunning():
    # Try to receive a frame
    frame = capture.getNextFrame()

    # Check if anything was received
    if frame is not None:
        # Pass the frame to the tracker
        tracker.accept(frame)

        # Run tracking
        result = tracker.runTracking()

```

(continues on next page)

(continued from previous page)

```
        # Pass tracking result into the track container which aggregates track history
        tracks.accept(result)

        # Generate and show a preview of recent tracking history
        cv.imshow("Preview", tracks.visualize(frame.image))

    cv.waitKey(2)
```

Fig. 5: Tracked features on a live frame from a camera.

4.3.2 Event-based tracking

Event-based Lucas Kanade tracker

Features can be detected and tracked on a stream of events. The *dv::features::EventFeatureLKTracker* can perform this, it accumulates a frame from events internally, runs feature detection and performs Lucas-Kanade tracking on the accumulated frames.

The following sample code shows how to use the event-only Lucas-Kanade tracker on event stream coming from a live camera.

C++

Python

```cpp
#include <dv-processing/features/event_feature_lk_tracker.hpp>
#include <dv-processing/features/feature_tracks.hpp>
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports event stream output, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    const cv::Size resolution = capture.getEventResolution().value();

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Instantiate a visual tracker with known resolution, all parameters kept default
    auto tracker = dv::features::EventFeatureLKTracker<>::RegularTracker(resolution);

    // Run tracking by accumulating frames with 100 FPS
    tracker->setFramerate(100);

    // Create a track container instance that is used to visualize tracks on an image
    dv::features::FeatureTracks tracks;

    // Run the frame processing while the camera is connected
    while (capture.isRunning()) {
        // Try to receive a batch of events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // Pass the frame to the tracker
            tracker->accept(*events);

            // Run tracking
            const auto result = tracker->runTracking();

            // Since we are passing events in fine-grained batches, tracking will not execute
            // until enough events is received, returning invalid pointer if tracking did not execute
            if (!result) {
                continue;
            }

            // Pass tracking result into the track container which aggregates track history
            tracks.accept(result);

            // Generate and show a preview of recent tracking history
            cv::imshow("Preview", tracks.visualize(tracker->getAccumulatedFrame()));
        }
        cv::waitKey(2);
```

(continues on next page)

(continued from previous page)

```
    }

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv

# Open any camera
capture = dv.io.CameraCapture()

# Make sure it supports event stream output, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Instantiate a visual tracker with known resolution, all parameters kept default
tracker = dv.features.EventFeatureLKTracker.RegularTracker(capture.
↪getEventResolution())

# Run tracking by accumulating frames with 100 FPS
tracker.setFramerate(100)

# Create a track container instance that is used to visualize tracks on an image
tracks = dv.features.FeatureTracks()

# Run the frame processing while the camera is connected
while capture.isRunning():
    # Try to receive a batch of events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # Pass the events to the tracker
        tracker.accept(events)

        # Run tracking
        result = tracker.runTracking()

        # Since we are passing events in fine-grained batches, tracking will not␣
↪execute
        # until enough events is received, returning a `None` if tracking did not␣
↪execute
        if result is None:
            continue

        # Pass tracking result into the track container which aggregates track history
        tracks.accept(result)

        # Generate and show a preview of recent tracking history
        cv.imshow("Preview", tracks.visualize(tracker.getAccumulatedFrame()))

    cv.waitKey(2)
```

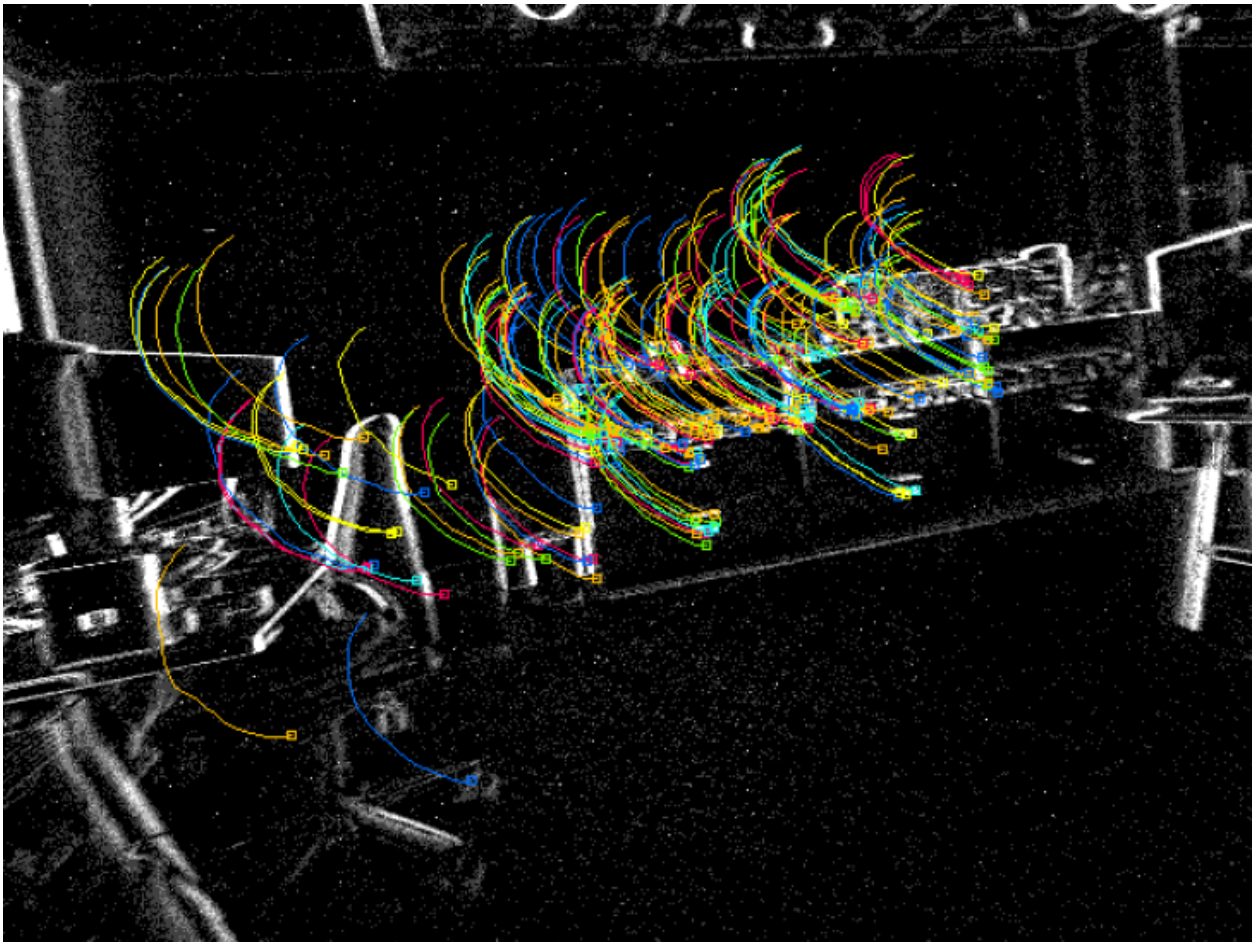


Fig. 6: Tracked features on a stream of events from a camera.

Event-based mean shift tracker

Detect and track features on a stream of events using mean shift algorithm. Although commonly used for clustering, the *dv::features::MeanShiftTracker* class provides a tracking implementation on event data based on mean shift update. The class internally detects interesting features to track from events (by default it uses *dv::features::EventBlobDetector*) and tracks them by running a mean shift update on a normalized time surface of events. The tracking is performed by following the interesting points detected on the time surface. The algorithm will shift the tracks towards the latest events, since it takes into account the intensity of the time surface when performing the track location update.

The algorithm can be summarized as follows:

1. Given a set of events, detect interesting blobs using *dv::features::EventBlobDetector*. (Note, this step happens if no track has been initialized or if redetection is enabled)
2. Compute the time surface representation of a given interval duration.
3. Given a set of input track locations, for each non-converged track retrieve the time surface of events within a configured window.
4. Calculate the mean of coordinates for the retrieved neighborhood, weighting each coordinate by the time surface intensity value.
5. Shift the initial track location by a mode, which is a vector going from the initial point to the mean coordinate multiplied by a learning rate factor.
6. If the mode of a vector is lower than a configured threshold, the track is considered to have converged into the new position, otherwise repeat from step one.

This algorithm is useful to track event blobs that could be used as point of interest in event processing algorithms.

The following code sample shows the use of our mean-shift tracker implementation to find and track events on sample data generated synthetically.

C++

Python

```cpp
#include <dv-processing/core/event.hpp>
#include <dv-processing/data/generate.hpp>
#include <dv-processing/features/mean_shift_tracker.hpp>
#include <dv-processing/visualization/events_visualizer.hpp>

#include <opencv2/highgui.hpp>
#include <opencv2/imgproc.hpp>

[[nodiscard]] dv::EventStore generateEventClustersAtTime(const int64_t time, const␣
↪std::vector<dv::Point2f> &clusters,
    const uint64_t numIter, const cv::Size &resolution, const int shift = -5);

int main() {
    using namespace std::chrono_literals;

    // Use VGA resolution
    const cv::Size resolution(640, 480);

    // Initialize a slicer
    dv::EventStreamSlicer slicer;

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);
```

(continues on next page)

(continued from previous page)

```

    // Initialize a list of clusters for synthetic data generation
    const std::vector<dv::Point2f> clusters(
        {dv::Point2f(550.f, 400.f), dv::Point2f(70.f, 300.f), dv::Point2f(305.f, 100.
↪f)});

    // Generate some random events for a background
    dv::EventStore events = dv::data::generate::uniformlyDistributedEvents(0,␣
↪resolution, 10'000);

    std::vector<int64_t> timestamps = {0, 40000, 80000, 120000, 160000, 200000,␣
↪240000, 280000, 320000, 360000};

    uint64_t numIter = 0;
    for (const auto time : timestamps) {
        auto eventCluster = generateEventClustersAtTime(time, clusters, numIter,␣
↪resolution);
        events            += eventCluster;
        events            += dv::data::generate::uniformlyDistributedEvents(time,␣
↪resolution, 10000, numIter);
        numIter++;
    }

    // Bandwidth value defining the size of the search window in which updated track␣
↪location will be searched
    const int bandwidth = 10;

    // Time window used for the normalized time surface computation. In this case we␣
↪take the last 50ms of events and
    // compute a normalized time surface over them
    const dv::Duration timeWindow = 50ms;

    // Initialize a mean shift tracker.
    dv::features::MeanShiftTracker meanShift =␣
↪dv::features::MeanShiftTracker(resolution, bandwidth, timeWindow);

    dv::visualization::EventVisualizer visualizer(resolution);

    // Register a callback every 40 milliseconds
    slicer.doEveryTimeInterval(40ms, [&](const dv::EventStore &events) {
        meanShift.accept(events);
        auto meanShiftTracks = meanShift.runTracking();

        if (!meanShiftTracks) {
            return;
        }

        // visualize mean shift tracks
        auto preview = visualizer.generateImage(events);
        auto points  = dv::data::fromTimedKeyPoints(meanShiftTracks->keypoints);
        cv::drawKeypoints(preview, points, preview, dv::visualization::colors::red);

        cv::imshow("Preview", preview);
        cv::waitKey(300);
    });

    slicer.accept(events);
```

(continues on next page)

(continued from previous page)

```cpp

    return EXIT_SUCCESS;
}

dv::EventStore generateEventClustersAtTime(const int64_t time, const std::vector
↪<dv::Point2f> &clusters,
    const uint64_t numIter, const cv::Size &resolution, const int shift) {
    // Declare a region filter which we will use to filter out-of-bounds events in␣
↪the next step
    dv::EventRegionFilter filter(cv::Rect(0, 0, resolution.width, resolution.height));
    const float offset = static_cast<float>(shift * static_cast<int>(numIter));
    dv::EventStore eventFiltered;
    for (const auto &cluster : clusters) {
        const auto xShift       = cluster.x() + offset;
        const auto yShift       = cluster.y() + offset;
        const dv::Point2f point = dv::Point2f(xShift, yShift);
        // Generate a batch of normally distributed events around each of the cluster␣
↪centers
        filter.accept(dv::data::generate::normallyDistributedEvents(time, point,␣
↪dv::Point2f(3.f, 3.f), 1'000));

        // Apply region filter to the events to filter out events outside valid␣
↪dimensions
        eventFiltered += filter.generateEvents();
    }

    return eventFiltered;
}
```

```python
import datetime

import dv_processing as dv
import cv2 as cv


def generate_event_clusters_at_time(time, clusters, num_iter, shift=-5):
    # Declare a region filter which we will use to filter out-of-bounds events in the␣
↪next step
    event_filter = dv.EventRegionFilter((0, 0, resolution[0], resolution[1]))
    event_filtered = dv.EventStore()
    track_id = 0
    offset = shift * num_iter
    for cluster in clusters:
        x_coord = cluster[0] + offset
        y_coord = cluster[1] + offset

        # Generate a batch of normally distributed events around each of the cluster␣
↪centers
        event_filter.accept(dv.data.generate.normallyDistributedEvents(time, (x_coord,
↪ y_coord), (3, 3), 1000))

        # Apply region filter to the events to filter out events outside valid␣
↪dimensions
        event_filtered.add(event_filter.generateEvents())

        track_id += 1
```

(continues on next page)

(continued from previous page)

```
    return event_filtered


def run_mean_shift(events):
    mean_shift.accept(events)
    mean_shift_tracks = mean_shift.runTracking()

    preview = visualizer.generateImage(events)

    # Draw markers on each of the track coordinates
    if len(mean_shift_tracks.keypoints) > 0:
        for index in range(len(mean_shift_tracks.keypoints)):
            track = mean_shift_tracks.keypoints[index]
            cv.drawMarker(preview, (int(track.pt[0]), int(track.pt[1])), dv.visualization.colors.red(), cv.MARKER_CROSS,
                          20, 2)

    # Show the preview image with detected tracks
    cv.imshow("Preview", preview)
    cv.waitKey(10)


# Use VGA resolution
resolution = (640, 480)

# Initialize a slicer
slicer = dv.EventStreamSlicer()

# Initialize a preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Initialize a list of clusters for synthetic data generation
clusters = [(550, 400), (70, 300), (305, 100)]

# Generate some random events for a background
events = dv.data.generate.uniformlyDistributedEvents(0, resolution, 10000)

timestamps = [0, 40000, 80000, 120000, 160000, 200000, 240000, 280000, 320000, 360000]

num_iter = 0
for time in timestamps:
    event_cluster = generate_event_clusters_at_time(time, clusters, num_iter)
    events.add(event_cluster)
    events.add(dv.data.generate.uniformlyDistributedEvents(time, resolution, 10000, num_iter))
    num_iter += 1

# parameter defining the spatial window [pixels] in which the new track position will be searched
bandwidth = 10

# window of time used to compute the time surface used for the tracking update
time_window = datetime.timedelta(milliseconds=50)

# Initialize a mean shift tracker
mean_shift = dv.features.MeanShiftTracker(resolution, bandwidth, time_window)
```

(continues on next page)

(continued from previous page)

```

visualizer = dv.visualization.EventVisualizer(resolution)

slicer.doEveryTimeInterval(datetime.timedelta(milliseconds=40), run_mean_shift)

slicer.accept(events)
```

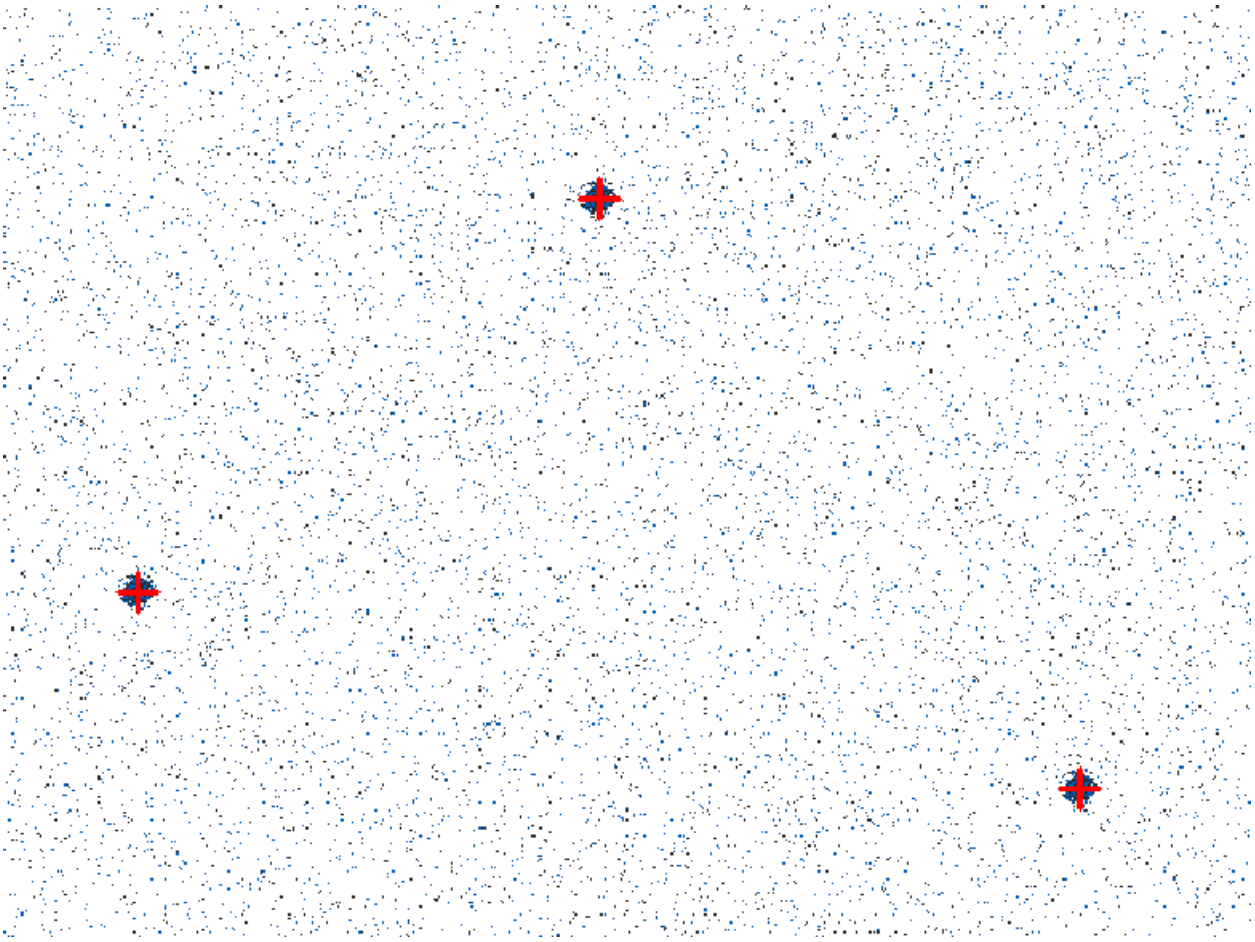


Fig. 7: Expected output of the mean-shift-tracker sample usage. Tracking eight blobs marked with red crosses.

4.3.3 Hybrid tracking

The high-framerate tracking on event stream suggests that the feature tracking on frames can be improved by tracking features between frames on intermediate accumulated frames from events. The intermediate tracking results can be used as a prior to the frame tracking algorithm. Such an approach is implemented in *dv::features::EventCombinedLK-Tracker*, it performs regular Lucas-Kanade tracking on frames, but also constructs intermediate accumulated frames to predict the locations of tracks in the next frame and uses this information as a prior to the Lucas-Kanade tracking algorithm.

The following sample code shows how to use the hybrid event-frame Lucas-Kanade tracker on both streams coming from a live camera.

Note: This sample requires a camera that is capable of producing frames and events, e.g. a DAVIS series camera.

C++

Python

```cpp
#include <dv-processing/features/event_combined_lk_tracker.hpp>
#include <dv-processing/features/feature_tracks.hpp>
#include <dv-processing/io/camera_capture.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Open any camera
    dv::io::CameraCapture capture;

    // Make sure it supports correct stream outputs, throw an error otherwise
    if (!capture.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }
    if (!capture.isFrameStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide a frame stream.");
    }

    const cv::Size resolution = capture.getEventResolution().value();

    // Initialize a preview window
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Instantiate a visual tracker with known resolution, all parameters kept default
    auto tracker = dv::features::EventCombinedLKTracker<>::RegularTracker(resolution);

    // Accumulate and track on 5 intermediate accumulated frames between each actual frame pair
    tracker->setNumIntermediateFrames(5);

    // Create a track container instance that is used to visualize tracks on an image
    dv::features::FeatureTracks tracks;

    // Use a queue to store incoming frames to make sure the all data has arrived prior to running the tracking
    std::queue<dv::Frame> frameQueue;

    // Run the frame processing while the camera is connected
    while (capture.isRunning()) {
        // Try to receive a frame, check if anything was received
        if (const auto frame = capture.getNextFrame()) {
            // Push the received frame into the frame queue
            frameQueue.push(*frame);
        }

        // Try to receive a batch of events, check if anything was received
        if (const auto events = capture.getNextEventBatch()) {
            // Pass the frame to the tracker
            tracker->accept(*events);
```

(continues on next page)

(continued from previous page)

```cpp

            // Check if we have ready frames and if enough events have arrived already
            if (frameQueue.empty() || frameQueue.front().timestamp > events->
↪getHighestTime()) {
                continue;
            }

            // Take the last frame from the queue
            const auto frame = frameQueue.front();

            // Pass it to the tracker as well
            tracker->accept(frame);

            // Remove the last used frame from the queue
            frameQueue.pop();

            // Run tracking
            const auto result = tracker->runTracking();

            // Validate that the tracking was successful
            if (!result) {
                continue;
            }

            // Pass tracking result into the track container which aggregates track␣
↪history
            tracks.accept(result);

            // Generate and show a preview of recent tracking history on both␣
↪accumulated frames and the frame image
            // Take the set of intermediate accumulated frames from the tracker
            const auto accumulatedFrames = tracker->getAccumulatedFrames();
            if (!accumulatedFrames.empty()) {
                cv::Mat preview;
                // Draw visualization on both image and concatenate them horizontally
                cv::hconcat(
                    tracks.visualize(accumulatedFrames.back().pyramid.front()),␣
↪tracks.visualize(frame.image), preview);
                // Show the final preview image
                cv::imshow("Preview", preview);
            }
        }

        cv::waitKey(2);
    }

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Open any camera
capture = dv.io.CameraCapture()

```

(continues on next page)

(continued from previous page)

```
# Make sure it supports correct stream outputs, throw an error otherwise
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")
if not capture.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide a frame stream.")

# Initialize preview window
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Instantiate a visual tracker with known resolution, all parameters kept default
tracker = dv.features.EventCombinedLKTracker.RegularTracker(capture.
↪getEventResolution())

# Accumulate and track on 5 intermediate accumulated frames between each actual frame␣
↪pair
tracker.setNumIntermediateFrames(5)

# Create a track container instance that is used to visualize tracks on an image
tracks = dv.features.FeatureTracks()

# Use a list to store incoming frames to make sure the all data has arrived prior to␣
↪running the tracking
frame_queue = []

# Run the frame processing while the camera is connected
while capture.isRunning():
    # Try to receive a frame
    frame = capture.getNextFrame()

    # Check if anything was received
    if frame is not None:
        frame_queue.append(frame)

    # Try to receive a batch of events
    events = capture.getNextEventBatch()

    # Check if anything was received
    if events is not None:
        # Pass the events to the tracker
        tracker.accept(events)

        # Check if we have ready frames and if enough events have arrived already
        if len(frame_queue) == 0 or frame_queue[0].timestamp > events.
↪getHighestTime():
            continue

        # Take the last frame from the queue and remove it
        frame = frame_queue.pop(0)

        # Pass it to the tracker as well
        tracker.accept(frame)

        # Run tracking
        result = tracker.runTracking()

        # Validate that the tracking was successful
        if result is None:
```

(continues on next page)

(continued from previous page)

```
            continue

        # Pass tracking result into the track container which aggregates track history
        tracks.accept(result)

        # Generate and show a preview of recent tracking history on both accumulated frames and the frame image
        # Take the set of intermediate accumulated frames from the tracker
        accumulated_frames = tracker.getAccumulatedFrames()
        if len(accumulated_frames) > 0:
            # Draw visualization on both image and concatenate them horizontally
            preview = cv.hconcat([tracks.visualize(accumulated_frames[-1].pyramid[0]), tracks.visualize(frame.image)])

            # Show the final preview image
            cv.imshow("Preview", preview)

    cv.waitKey(2)
```

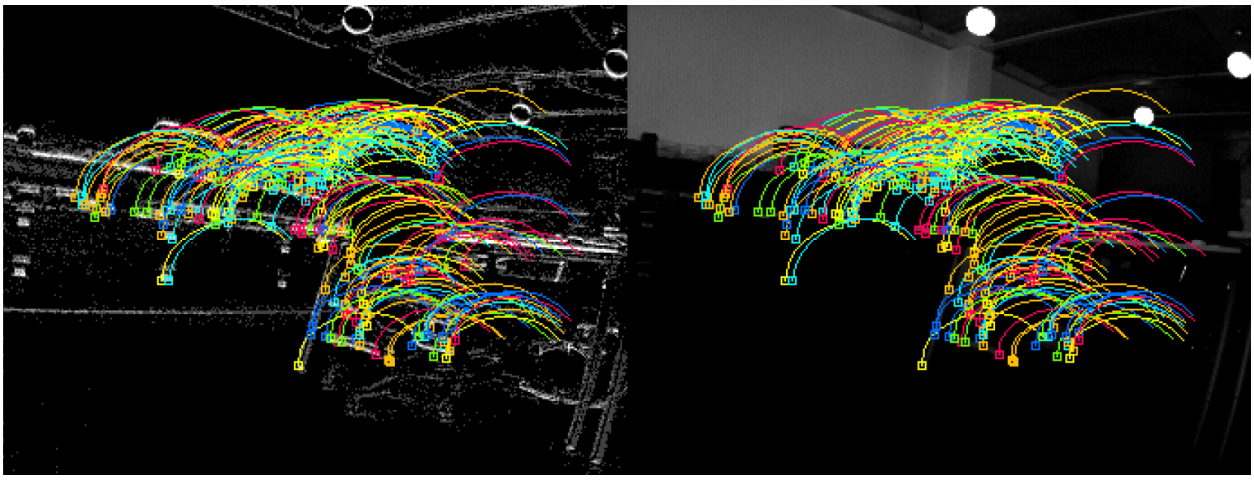


Fig. 8: Tracked features on frame and event streams from a camera.

4.4 Kinematics primitives

The dv-processing library provides a minimal implementation of kinematic transformation primitives that are useful for geometric computer vision algorithms. This includes transformations of 3D points using transformation matrices and handling rigid-body motion trajectories over time. These basic primitives can be used to implement motion compensation algorithm for event data, which can be used to reduce or eliminate motion blur in the event stream. The underlying implementation uses mathematical operations from Eigen library, so the mathematical operations are expected to be highly efficient.

4.4.1 Transformation

A transformation in dv-processing describes an object's orientation and position in 3D space at a certain point in time. Transformation contains a timestamp, rotational and translational transformation expressed as a 4x4 homogenous transformation matrix T, like this:

$$T = \begin{bmatrix} r_0 & r_1 & r_2 & t_0 \\ r_3 & r_4 & r_5 & t_1 \\ r_6 & r_7 & r_8 & t_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} r & t \\ 0 & 1 \end{bmatrix}$$

Here:

- $r, r_{0..9}$ - is a rotation matrix (and it's coefficients) that describes an object's rotation.
- $t, t_{0..2}$ - is a vector describing translational vector, which is an object's displacement.

A transformation matrix together with a timestamp describes complete attitude with 6 degrees of freedom. This transformation can be applied to other transformations as well as 3D points to obtain a new relative position with the applied transformation. The transformation is implemented in the *dv::kinematics::Transformation* class, which is a templated class. The template parameter sets the underlying matrix scalar data type for the 4x4 matrix, which is either `float` or `double`. To simplify the use case, two predefined aliases are defined: *dv::kinematics::Transformationf* and *dv::kinematics::Transformationd* - they differ in the underlying scalar data type:

- `Transformationf` uses 32-bit single precision floating point values,
- `Transformationd` uses 64-bit double precision floating point values.

The library usually prefers the use of single precision floating point scalar type, since the representation is accurate enough for sub-millimeter accuracy with lower memory footprint.

The following sample code shows how to initialize a transformation and apply it to a 3D point.

C++

Python

```cpp
#include <dv-processing/kinematics/transformation.hpp>

#include <iostream>

int main() {
    Eigen::Matrix4f matrix;

    // Mirror rotation matrix with 0.5 translational offsets on all axes. The rotation matrix should flip
    // x and z axes of the input.
    matrix << -1.f, 0.f, 0.f, 0.5f, 0.f, 1.f, 0.f, 0.5f, 0.f, 0.f, -1.f, 0.5f, 0.f, 0.f, 0.f, 1.f;

    // Initialize the transformation with the above matrix. The timestamp can be ignored for this sample, so its set
    // to zero.
    const dv::kinematics::Transformationf transformation(0, matrix);

    // Let's take a sample point with offsets of 1 on all axes.
    const Eigen::Vector3f point(1.f, 1.f, 1.f);

    // Apply this transformation to the above point. This should invert x and z axes and add 0.5 to all values.
```

(continues on next page)

(continued from previous page)

```cpp
    const Eigen::Vector3f transformed = transformation.transformPoint(point);

    // Print the resulting output.
    std::cout << "Transformed from [" << point.transpose() << "] to [" << transformed.transpose() << "]" << std::endl;

    return 0;
}
```

```python
import dv_processing as dv
import numpy as np

# Mirror rotation matrix with 0.5 translational offsets on all axes. The rotation matrix should flip
# x and z axes of the input.
matrix = np.array([[-1.0, 0.0, 0.0, 0.5], [0.0, 1.0, 0.0, 0.5], [0.0, 0.0, -1.0, 0.5], [0.0, 0.0, 0.0, 1.]])

# Initialize the transformation with the above matrix. The timestamp can be ignored for this sample, so its set
# to zero.
transformation = dv.kinematics.Transformationf(0, matrix)

# Let's take a sample point with offsets of 1 on all axes.
point = np.array([1.0, 1.0, 1.0])

# Apply this transformation to the above point. This should invert x and z axes and add 0.5 to all values.
transformed = transformation.transformPoint(point)

# Print the resulting output.
print(f"Transformed from {point} to {transformed}")
```

4.4.2 Linear transformer

A set of transformations that are monotonically increasing in time can be formed into a motion trajectory. Linear transformer can be used to store a set of transformation representing a single objects trajectory and extract transformations at specified points in time, which are calculated using linear interpolation between the nearest available transformations.

The following sample code shows how to use the *dv::kinematics::LinearTransformerf* class to interpolate intermediate transformations in time:

C++

Python

```cpp
#include <dv-processing/kinematics/linear_transformer.hpp>
#include <dv-processing/kinematics/transformation.hpp>

#include <iostream>

int main() {
    // Declare linear transformer with capacity of 100 transformations. Internally it uses a bounded FIFO queue
    // to manage the transformations.
```

(continues on next page)

(continued from previous page)

```cpp
    dv::kinematics::LinearTransformerf transformer(100);

    // Push first transformation which is an identity matrix, so it starts with no rotation at zero coordinates
    transformer.pushTransformation(
        dv::kinematics::Transformationf(1000000, Eigen::Vector3f(0.f, 0.f, 0.f), Eigen::Quaternionf::Identity()));

    // Add a second transformation with no rotation as well, but with different translational coordinates
    transformer.pushTransformation(
        dv::kinematics::Transformationf(2000000, Eigen::Vector3f(1.f, 2.f, 3.f), Eigen::Quaternionf::Identity()));

    // Interpolate transformation at a midpoint (time-wise), this should device the translational coordinates
    // by a factor of 2.0
    const auto midpoint = transformer.getTransformAt(1500000);

    // Print the resulting output.
    std::cout << "Interpolated position at [" << midpoint->getTimestamp() << "]: ["
              << midpoint->getTranslation().transpose() << "]" << std::endl;

    return 0;
}
```

```python
import dv_processing as dv
import numpy as np

# Declare linear transformer with capacity of 100 transformations. Internally it uses a bounded FIFO queue
# to manage the transformations.
transformer = dv.kinematics.LinearTransformerf(100)

# Push first transformation which is an identity matrix, so it starts with no rotation at zero coordinates
transformer.pushTransformation(dv.kinematics.Transformationf(1000000, np.array([0.0, 0.0, 0.0]), (1.0, 0.0, 0.0, 0.0)))

# Add a second transformation with no rotation as well, but with different translational coordinates
transformer.pushTransformation(dv.kinematics.Transformationf(2000000, np.array([1.0, 2.0, 3.0]), (1.0, 0.0, 0.0, 0.0)))

# Interpolate transformation at a midpoint (time-wise), this should device the translational coordinates
# by a factor of 2.0
midpoint = transformer.getTransformAt(1500000)

# Print the resulting output.
print(f"Interpolated position at [{midpoint.getTimestamp()}]: {midpoint.getTranslation()}")
```

4.5 Mean-shift clustering

Mean-shift clustering algorithm is used to find high density clusters of events (or point) within a set of events. The dv-processing library provides an implementation of such an algorithm which is efficiently implemented for incoming streams of events from event cameras.

The algorithm can be summarized as follows:

1. Given a set of coordinates, for each coordinate retrieve a neighborhood of events within a configured window.
2. Calculate the mean of coordinates for the retrieved neighborhood.
3. Shift the initial coordinates by a mode, which is a vector going from the initial point to the mean coordinate multiplied by a learning rate factor.
4. If the mode of a vector is lower than a configured threshold, the coordinate is considered to have converged into a cluster center, otherwise repeat from step one.

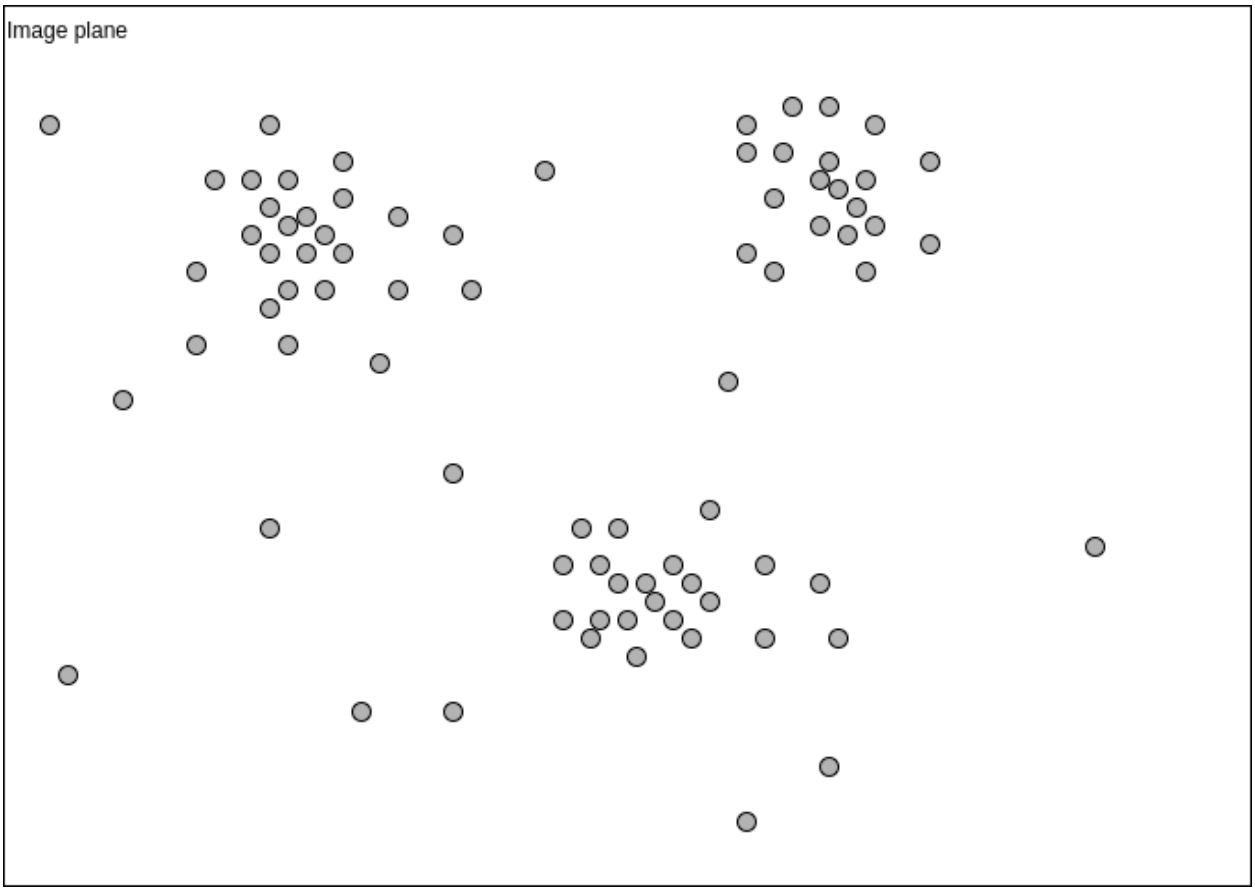


Fig. 9: An approximate visualization of a mean-shift clustering algorithm to find centers of coordinate clusters in a 2D plane.

This algorithm is useful to find event clusters that could be used as point of interest in event processing algorithms.

4.5.1 Sample usage

The following code sample shows the use of mean-shift clustering implementation to find and track clusters of events on simulated events.

C++

Python

```cpp
#include <dv-processing/cluster/mean_shift/event_store_adaptor.hpp>
#include <dv-processing/core/event.hpp>
#include <dv-processing/data/generate.hpp>
#include <dv-processing/visualization/events_visualizer.hpp>

#include <opencv2/highgui.hpp>

int main() {
    // Use VGA resolution
    const cv::Size resolution(640, 480);

    // Initialize a list of clusters for synthetic data generation
    const std::vector<dv::Point2f> clusters(
        {dv::Point2f(100.f, 100.f), dv::Point2f(462.f, 25.f), dv::Point2f(105.f, 340.
↪f), dv::Point2f(540.f, 420.f)});

    // Generate some random events for a background
    dv::EventStore events = dv::data::generate::uniformlyDistributedEvents(0,␣
↪resolution, 10'000);

    // Declare a region filter which we will use to filter out-of-bounds events in␣
↪the next step
    dv::EventRegionFilter filter(cv::Rect(0, 0, resolution.width, resolution.height));

    for (const auto &cluster : clusters) {
        // Generate a batch of normally distributed events around each of the cluster␣
↪centers
        filter.accept(dv::data::generate::normallyDistributedEvents(0, cluster,␣
↪dv::Point2f(15.f, 15.f), 5'000));

        // Apply region filter to the events to filter out events outside valid␣
↪dimensions
        events += filter.generateEvents();
    }

    // Initialize mean shift clustering algorithm, with initial parameters of:
    // bandwidth = 100, this is pixel search radius around a point
    // conv = 0.01, the search converges when the magnitude of mean-shift vector is␣
↪below this value
    // maxIter = 10000, maximum number of mean-shift update iterations
    // numStartingPoints = 100, number of randomly selected starting points
    dv::cluster::mean_shift::MeanShiftEventStoreAdaptor meanShift(events, 100.0f, 0.
↪01f, 10000, 100);

    // Perform the mean-shift, the algorithm returns a tuple of center coordinates,␣
↪labels, count of event in
    // the cluster, and variances of the cluster
    auto [centers, labels, counts, variances] = meanShift.fit();

```

(continues on next page)

(continued from previous page)

```cpp
    // Let's assign the cluster size to the response value of the center keypoint
    for (int i = 0; i < centers.size(); i++) {
        centers[i].response = static_cast<float>(counts[i]);
    }

    // Sort the estimated centers by the number of events in cluster, the values are sorted in descending order
    std::sort(centers.begin(), centers.end(), [](const auto &a, const auto &b) {
        return a.response > b.response;
    });

    // Choose top four center with most events; these centers should be close to initial hardcoded cluster centers
    if (centers.size() > 4) {
        centers.resize(4);
    }

    // Use event visualizer to generate a preview image
    dv::visualization::EventVisualizer visualizer(resolution);
    auto preview = visualizer.generateImage(events);

    // Draw markers on each of the center coordinates
    for (const auto &center : centers) {
        cv::drawMarker(preview, cv::Point2f(center.pt.x(), center.pt.y()), dv::visualization::colors::red);
    }

    // Show the preview image with detected cluster centers
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);
    cv::imshow("Preview", preview);
    cv::waitKey();

    return 0;
}
```

```python
import dv_processing as dv
import cv2 as cv

# Use VGA resolution
resolution = (640, 480)

# Initialize a list of clusters for synthetic data generation
clusters = [(100, 100), (462, 25), (105, 340), (540, 420)]

# Generate some random events for a background
events = dv.data.generate.uniformlyDistributedEvents(0, resolution, 10000)

# Declare a region filter which we will use to filter out-of-bounds events in the next step
filter = dv.EventRegionFilter((0, 0, resolution[0], resolution[1]))

for cluster in clusters:
    # Generate a batch of normally distributed events around each of the cluster centers
    filter.accept(dv.data.generate.normallyDistributedEvents(0, cluster, (15, 15), 5000))
```

(continues on next page)

(continued from previous page)

```
    # Apply region filter to the events to filter out events outside valid dimensions
    events.add(filter.generateEvents())

# Initialize mean shift clustering algorithm, with initial parameters of:
# bandwidth = 100, this is pixel search radius around a point
# conv = 0.01, the search converges when the magnitude of mean-shift vector is below this value
# maxIter = 10000, maximum number of mean-shift update iterations
# numStartingPoints = 100, number of randomly selected starting points
mean_shift = dv.cluster.mean_shift.MeanShiftEventStoreAdaptor(events, 100, 0.01, 10000, 100)

# Perform the mean-shift, the algorithm returns a tuple of center coordinates, labels, count of event in the
# cluster, and variances of the cluster
centers, labels, counts, variances = mean_shift.fit()

# Let's assign the cluster size to the response value of the center keypoint
for i in range(len(centers)):
    centers[i].response = counts[i]

# Sort the estimated centers by the number of events in cluster, the values are sorted in descending order
centers.sort(key=lambda a: a.response, reverse=True)

# Choose top four center with most events; these centers should be close to initial hardcoded cluster centers
if len(centers) > 4:
    centers = centers[:4]

# Use event visualizer to generate a preview image
visualizer = dv.visualization.EventVisualizer(resolution)
preview = visualizer.generateImage(events)

# Draw markers on each of the center coordinates
for center in centers:
    cv.drawMarker(preview, (int(center.pt[0]), int(center.pt[1])), dv.visualization.colors.red())

# Show the preview image with detected cluster centers
cv.namedWindow("Preview", cv.WINDOW_NORMAL)
cv.imshow("Preview", preview)
cv.waitKey()
```

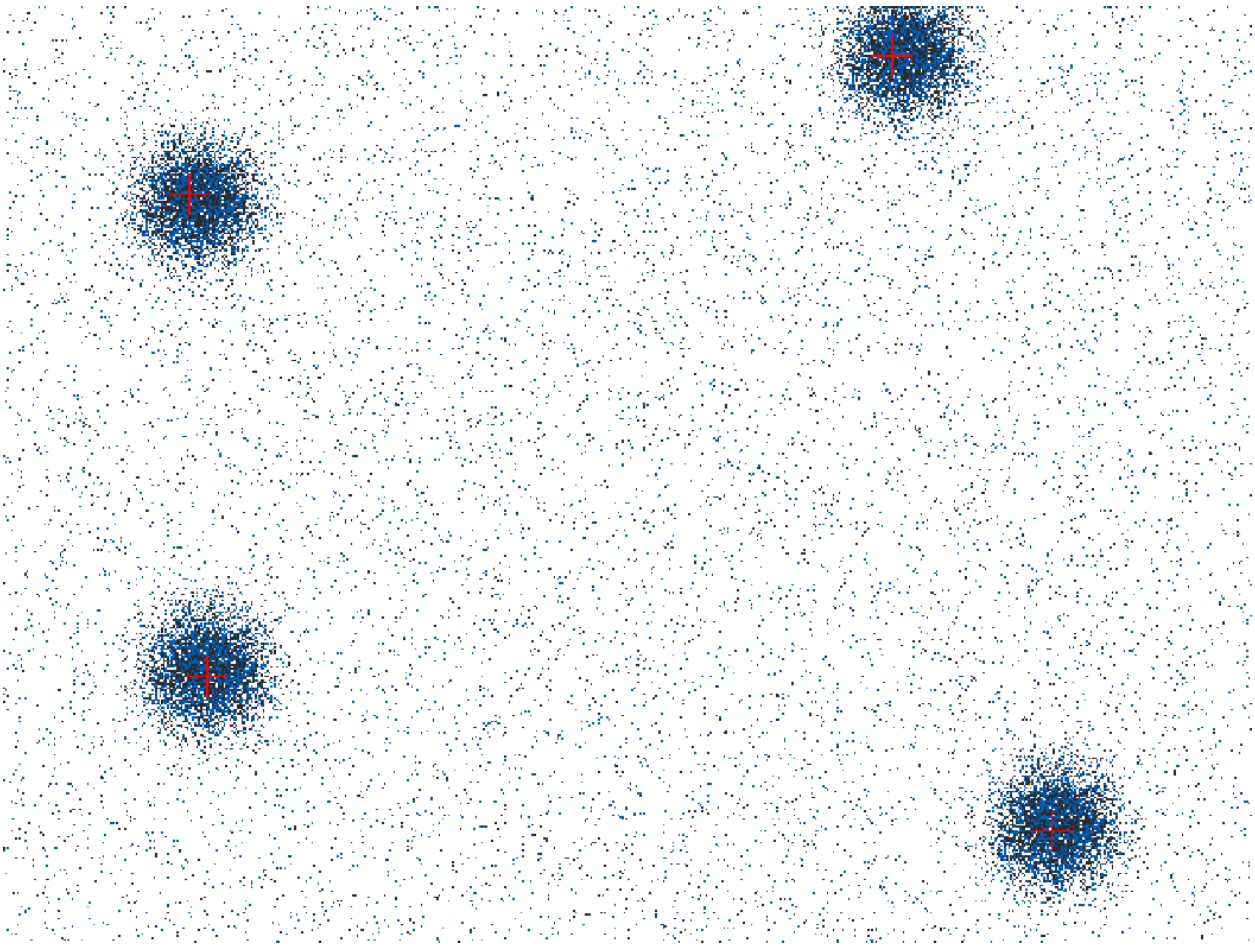


Fig. 10: Expected output of the mean-shift sample usage, four cluster centers marked with red crosses.

CHAPTER

FIVE

ADVANCED APPLICATIONS

This section describes the use of available library features and algorithms to build more complex applications and algorithms for: depth estimation, motion compensation and contrast maximization.

5.1 Depth estimation

Depth estimation with event cameras is possible by applying the same approach of disparity calculation on a calibrated stereo camera rig. The straightforward approach is to accumulate frames from events on both cameras and use the same disparity estimation algorithm. This approach might have some limitations, since accumulating events might result in suboptimal results due to low texture available in an accumulated frame.

The dv-processing library provides the *dv::camera::StereoGeometry* and a few disparity estimation algorithms that, in combination, can be used to build a depth estimation pipeline.

5.1.1 Semi-dense stereo block matching

Dense block matching here refers to the most straightforward approach: accumulating full frames and running a conventional disparity estimation on top to estimate depth. Since the accumulated frames only contain limited texture due to pixels reacting to brightness changes - this approach is referred to as semi-dense. The `SemiDenseStereoMatcher` class wraps the disparity estimation part, where estimated disparity can be used to calculate depth with *dv::camera::StereoGeometry*.

Following sample code show the use of `SemiDenseStereoMatcher` with *dv::camera::StereoGeometry* to run a real-time depth estimation pipeline on a calibration stereo camera.

C++

Python

```cpp
#include <dv-processing/camera/calibration_set.hpp>
#include <dv-processing/core/stereo_event_stream_slicer.hpp>
#include <dv-processing/depth/semi_dense_stereo_matcher.hpp>
#include <dv-processing/io/stereo_capture.hpp>
#include <dv-processing/noise/background_activity_noise_filter.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Path to a stereo calibration file, replace with a file path on your local file␣
↪system
```

(continues on next page)

(continued from previous page)

```
    const std::string calibrationFilePath = "path/to/calibration.json";

    // Load the calibration file
    auto calibration = dv::camera::CalibrationSet::LoadFromFile(calibrationFilePath);

    // It is expected that calibration file will have "C0" as the leftEventBuffer camera
    auto leftCamera = calibration.getCameraCalibration("C0").value();

    // The second camera is assumed to be rightEventBuffer-side camera
    auto rightCamera = calibration.getCameraCalibration("C1").value();

    // Open the stereo camera with camera names from calibration
    dv::io::StereoCapture capture(leftCamera.name, rightCamera.name);

    // Make sure both cameras support event stream output, throw an error otherwise
    if (!capture.left.isEventStreamAvailable() || !capture.right.isEventStreamAvailable()) {
        throw dv::exceptions::RuntimeError("Input camera does not provide an event stream.");
    }

    // Initialize a stereo block matcher with a stereo geometry from calibration and the preconfigured SGBM instance
    dv::SemiDenseStereoMatcher blockMatcher(std::make_unique<dv::camera::StereoGeometry>(leftCamera, rightCamera));

    // Initialization of a stereo event sliver
    dv::StereoEventStreamSlicer slicer;

    // Initialize a window to show previews of the output
    cv::namedWindow("Preview", cv::WINDOW_NORMAL);

    // Local event buffers to implement overlapping window of events for accumulation
    dv::EventStore leftEventBuffer, rightEventBuffer;

    // Use one third of the resolution as count of events per accumulated frame
    const size_t eventCount = static_cast<size_t>(leftCamera.resolution.area()) / 3;

    // Register a callback to be done at 30Hz
    slicer.doEveryTimeInterval(33ms, [&blockMatcher, &leftEventBuffer, &rightEventBuffer, eventCount](
                                         const auto &leftEvents, const auto &rightEvents) {
        // Push input events into the local buffers
        leftEventBuffer.add(leftEvents);
        rightEventBuffer.add(rightEvents);

        // If the number of events is above the count, just keep the latest events
        if (leftEventBuffer.size() > eventCount) {
            leftEventBuffer = leftEventBuffer.sliceBack(eventCount);
        }
        if (rightEventBuffer.size() > eventCount) {
            rightEventBuffer = rightEventBuffer.sliceBack(eventCount);
        }

        // Pass these events into block matcher and estimate disparity, the matcher
```

(continues on next page)

(continued from previous page)

```
↪will accumulate frames
63          // internally. The disparity output is 16-bit integer, that has sub-pixel␣
↪precision.
64          const auto disparity = blockMatcher.computeDisparity(leftEventBuffer,␣
↪rightEventBuffer);
65
66          // Convert disparity into 8-bit integers with scaling and normalize the␣
↪output for a nice preview.
67          // This loses the actual numeric value of the disparity, but it's a nice way␣
↪to visualize the disparity.
68          cv::Mat disparityU8, disparityColored;
69          disparity.convertTo(disparityU8, CV_8UC1, 1.0 / 16.0);
70          cv::normalize(disparityU8, disparityU8, 0, 255, cv::NORM_MINMAX);
71
72          // Convert the accumulated frames into colored images for preview.
73          std::vector<cv::Mat> images(3);
74          cv::cvtColor(blockMatcher.getLeftFrame().image, images[0], cv::COLOR_
↪GRAY2BGR);
75          cv::cvtColor(blockMatcher.getRightFrame().image, images[1], cv::COLOR_
↪GRAY2BGR);
76
77          // Apply color-mapping to the disparity image, this will encode depth with␣
↪color: red - close; blue - far.
78          cv::applyColorMap(disparityU8, images[2], cv::COLORMAP_JET);
79
80          // Concatenate images and show them in a window
81          cv::Mat preview;
82          cv::hconcat(images, preview);
83          cv::imshow("Preview", preview);
84      });
85
86      // Buffer input events in these variables to synchronize inputs
87      std::optional<dv::EventStore> leftEvents  = std::nullopt;
88      std::optional<dv::EventStore> rightEvents = std::nullopt;
89
90      // Run the processing loop while both cameras are connected
91      while (capture.left.isRunning() && capture.right.isRunning()) {
92          // Read events from respective left / right cameras
93          if (!leftEvents.has_value()) {
94              leftEvents = capture.left.getNextEventBatch();
95          }
96          if (!rightEvents.has_value()) {
97              rightEvents = capture.right.getNextEventBatch();
98          }
99
100         // Feed the data into the slicer and reset the buffer
101         if (leftEvents && rightEvents) {
102             slicer.accept(*leftEvents, *rightEvents);
103             leftEvents  = std::nullopt;
104             rightEvents = std::nullopt;
105         }
106
107         // Wait for a small amount of time to avoid CPU overhaul
108         cv::waitKey(1);
109     }
110
111     return 0;
```

(continues on next page)

(continued from previous page)

```
}
```

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Path to a stereo calibration file, replace with a file path on your local file system
calibration_file_path = "path/to/calibration.json"

# Load the calibration file
calibration = dv.camera.CalibrationSet.LoadFromFile(calibration_file_path)

# It is expected that calibration file will have "C0" as the leftEventBuffer camera
left_camera = calibration.getCameraCalibration("C0")

# The second camera is assumed to be rightEventBuffer-side camera
right_camera = calibration.getCameraCalibration("C1")

# Open the stereo camera with camera names from calibration
capture = dv.io.StereoCapture(left_camera.name, right_camera.name)

# Make sure both cameras support event stream output, throw an error otherwise
if not capture.left.isEventStreamAvailable() or not capture.right.isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Initialize a stereo block matcher with a stereo geometry from calibration and the preconfigured SGBM instance
block_matcher = dv.SemiDenseStereoMatcher(dv.camera.StereoGeometry(left_camera, right_camera))

# Initialization of a stereo event sliver
slicer = dv.StereoEventStreamSlicer()

# Initialize a window to show previews of the output
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Local event buffers to implement overlapping window of events for accumulation
global left_event_buffer, right_event_buffer
left_event_buffer = dv.EventStore()
right_event_buffer = dv.EventStore()

# Use one third of the resolution as count of events per accumulated frame
event_count = int((left_camera.resolution[0] * left_camera.resolution[1]) / 3)


# Stereo slicer callback method
def callback(left_events: dv.EventStore, right_events: dv.EventStore):
    # Push input events into the local buffers
    global left_event_buffer, right_event_buffer
    left_event_buffer.add(left_events)
    right_event_buffer.add(right_events)

    # If the number of events is above the count, just keep the latest events
    if len(left_event_buffer) > event_count:
```

(continues on next page)

(continued from previous page)

```
        left_event_buffer = left_event_buffer.sliceBack(event_count)
    if len(right_event_buffer) > event_count:
        right_event_buffer = right_event_buffer.sliceBack(event_count)

    # Pass these events into block matcher and estimate disparity, the matcher will accumulate frames
    # internally. The disparity output is 16-bit integer, that has sub-pixel precision.
    disparity = block_matcher.computeDisparity(left_event_buffer, right_event_buffer)

    # Convert the accumulated frames into colored images for preview.
    images = []
    images.append(cv.cvtColor(block_matcher.getLeftFrame().image, cv.COLOR_GRAY2BGR))
    images.append(cv.cvtColor(block_matcher.getRightFrame().image, cv.COLOR_GRAY2BGR))

    # Convert disparity into 8-bit integers with scaling and normalize the output for a nice preview.
    # This loses the actual numeric value of the disparity, but it's a nice way to visualize the disparity.
    # Apply color-mapping to the disparity image, this will encode depth with color: red - close; blue - far.
    images.append(cv.applyColorMap(cv.normalize(disparity, None, 0, 255, cv.NORM_MINMAX, cv.CV_8UC1), cv.COLORMAP_JET))

    # Concatenate images and show them in a window
    cv.imshow("Preview", cv.hconcat(images))


# Register a callback to be done at 30Hz
slicer.doEveryTimeInterval(timedelta(milliseconds=33), callback)

# Buffer input events in these variables to synchronize inputs
left_events = None
right_events = None

# Run the processing loop while both cameras are connected
while capture.left.isRunning() and capture.right.isRunning():
    # Read events from respective left / right cameras
    if left_events is None:
        left_events = capture.left.getNextEventBatch()
    if right_events is None:
        right_events = capture.right.getNextEventBatch()

    # Feed the data into the slicer and reset the buffer
    if left_events is not None and right_events is not None:
        slicer.accept(left_events, right_events)
        left_events = None
        right_events = None

    #  Wait for a small amount of time to avoid CPU overhaul
    cv.waitKey(1)
```

Note: Disparity map yields results only in areas with visible texture, areas without texture contain speckle noise.

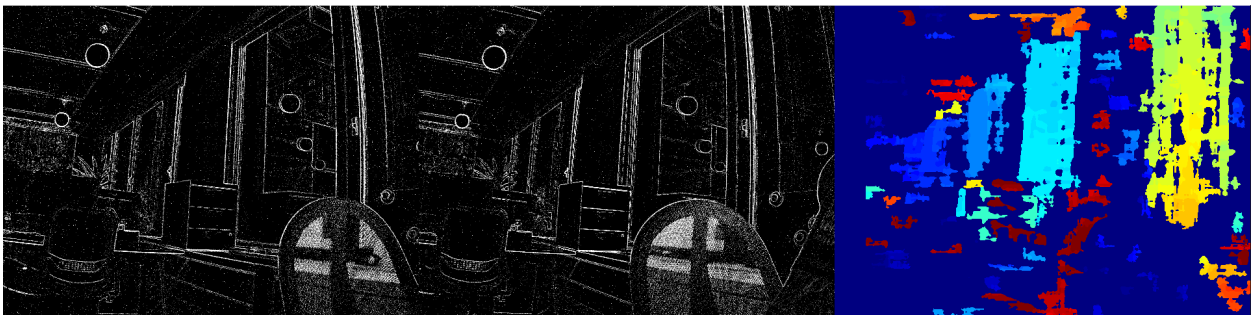


Fig. 1: Expected result of semi-dense disparity estimation. The output provides two accumulated frames and color-coded disparity map.

5.1.2 Sparse disparity estimation

The semi-dense appraoch is a straightforward to stereo disparity estimation. Another approach is to perform disparity estimation on sparse selected regions within accumulated image. Sparse estimation approach allows the implementation to select regions with enough texture to be selected for the disparity, reducing computational complexity and improving quality. The sparse approach takes point coordinates of where the disparity needs to be estimated, performs sparse accumulation only in the regions where disparity matching actually needs to happen and runs correlation based template matching of left image patches on the right camera image. Each template is matched against the other image on a horizontal line using normalized correlation coefficient (Pearson correlation) and the best scoring match is considered to be the correct match and according disparity is assigned to that point.

The following sample code shows the use of sparse disparity block matcher with a live calibrated stereo camera:

C++

Python

```cpp
#include <dv-processing/camera/calibration_set.hpp>
#include <dv-processing/cluster/mean_shift/event_store_adaptor.hpp>
#include <dv-processing/core/stereo_event_stream_slicer.hpp>
#include <dv-processing/data/utilities.hpp>
#include <dv-processing/depth/sparse_event_block_matcher.hpp>
#include <dv-processing/io/stereo_capture.hpp>
#include <dv-processing/visualization/colors.hpp>

#include <opencv2/highgui.hpp>

int main() {
    using namespace std::chrono_literals;

    // Path to a stereo calibration file, replace with a file path on your local file␣
↪system
    const std::string calibrationFilePath = "path/to/calibration.json";

    // Load the calibration file
    auto calibration = dv::camera::CalibrationSet::LoadFromFile(calibrationFilePath);

    // It is expected that calibration file will have "C0" as the leftEventBuffer␣
↪camera
    auto leftCamera = calibration.getCameraCalibration("C0").value();

    // The second camera is assumed to be rightEventBuffer-side camera
    auto rightCamera = calibration.getCameraCalibration("C1").value();
```

(continues on next page)

(continued from previous page)

```
25
26      // Open the stereo camera with camera names from calibration
27      dv::io::StereoCapture capture(leftCamera.name, rightCamera.name);
28
29      // Make sure both cameras support event stream output, throw an error otherwise
30      if (!capture.left.isEventStreamAvailable() || !capture.right.
    ↪isEventStreamAvailable()) {
31          throw dv::exceptions::RuntimeError("Input camera does not provide an event␣
    ↪stream.");
32      }
33
34      // Matching window size for the block matcher
35      const cv::Size window(24, 24);
36      // Minimum disparity value to measure
37      const int minDisparity = 0;
38      // Maximum disparity value
39      const int maxDisparity = 40;
40      // Minimum z-score value that a valid match can have
41      const float minScore = 0.0f;
42
43      // Initialize the block matcher with rectification
44      auto matcher = dv::SparseEventBlockMatcher(std::make_unique
    ↪<dv::camera::StereoGeometry>(leftCamera, rightCamera),
45          window, maxDisparity, minDisparity, minScore);
46
47      // Initialization of a stereo event sliver
48      dv::StereoEventStreamSlicer slicer;
49
50      // Initialize a window to show previews of the output
51      cv::namedWindow("Preview", cv::WINDOW_NORMAL);
52
53      // Local event buffers to implement overlapping window of events for accumulation
54      dv::EventStore leftEventBuffer, rightEventBuffer;
55
56      // Use one third of the resolution as count of events per accumulated frame
57      const size_t eventCount = static_cast<size_t>(leftCamera.resolution.area()) / 3;
58
59      // Register a callback to be done at 50Hz
60      slicer.doEveryTimeInterval(20ms, [&matcher, &leftEventBuffer, &rightEventBuffer,␣
    ↪eventCount, &window](
61                                         const auto &leftEvents, const auto &
    ↪rightEvents) {
62          // Push input events into the local buffers
63          leftEventBuffer.add(leftEvents);
64          rightEventBuffer.add(rightEvents);
65
66          // If the number of events is above the count, just keep the latest events
67          if (leftEventBuffer.size() > eventCount) {
68              leftEventBuffer = leftEventBuffer.sliceBack(eventCount);
69          }
70          if (rightEventBuffer.size() > eventCount) {
71              rightEventBuffer = rightEventBuffer.sliceBack(eventCount);
72          }
73
74          // Number of clusters to extract
75          constexpr int numClusters = 100;
76
```

(continues on next page)

(continued from previous page)

```
        // Initialize the mean-shift clustering algorithm
        dv::cluster::mean_shift::MeanShiftEventStoreAdaptor meanShift(leftEventBuffer,
↪ 10.f, 1.0f, 20, numClusters);

        // Find cluster centers which are going to be used for disparity estimation
        auto centers = meanShift.findClusterCentres<dv::cluster::mean_
↪shift::kernel::Epanechnikov>();

        // Run disparity estimation, the output will contain a disparity estimate for␣
↪each of the given points.
        const std::vector<dv::SparseEventBlockMatcher::PixelDisparity> estimates
            = matcher.computeDisparitySparse(leftEventBuffer, rightEventBuffer,␣
↪dv::data::convertToCvPoints(centers));

        // Convert the accumulated frames into colored images for preview.
        std::vector<cv::Mat> images(2);
        cv::cvtColor(matcher.getLeftFrame().image, images[0], cv::COLOR_GRAY2BGR);
        cv::cvtColor(matcher.getRightFrame().image, images[1], cv::COLOR_GRAY2BGR);

        // Visualize the matched blocks
        int32_t index = 0;
        for (const auto &point : estimates) {
            // If point estimation is invalid, do not show a preview of it
            if (!point.valid) {
                continue;
            }

            // The rest of the code performs drawing of the match according to the␣
↪disparity value on the
            // preview images.
            const cv::Scalar color =␣
↪dv::visualization::colors::someNeonColor(index++);
            // Draw some nice colored markers and rectangles.
            cv::drawMarker(images[1], *point.matchedPosition, color, cv::MARKER_CROSS,
↪ 7);
            cv::rectangle(images[1],
                cv::Rect(point.matchedPosition->x - (window.width / 2), point.
↪matchedPosition->y - (window.height / 2),
                    window.width, window.height),
                color);
            cv::rectangle(images[0],
                cv::Rect(point.templatePosition->x - (window.width / 2),
                    point.templatePosition->y - (window.height / 2), window.width,␣
↪window.height),
                color);
        }

        // Concatenate images and show them in a window
        cv::Mat preview;
        cv::hconcat(images, preview);
        cv::imshow("Preview", preview);
    });

    // Buffer input events in these variables to synchronize inputs
    std::optional<dv::EventStore> leftEvents  = std::nullopt;
    std::optional<dv::EventStore> rightEvents = std::nullopt;

```

(continues on next page)

(continued from previous page)

```
    // Run the processing loop while both cameras are connected
    while (capture.left.isRunning() && capture.right.isRunning()) {
        // Read events from respective left / right cameras
        if (!leftEvents.has_value()) {
            leftEvents = capture.left.getNextEventBatch();
        }
        if (!rightEvents.has_value()) {
            rightEvents = capture.right.getNextEventBatch();
        }

        // Feed the data into the slicer and reset the buffer
        if (leftEvents && rightEvents) {
            slicer.accept(*leftEvents, *rightEvents);
            leftEvents  = std::nullopt;
            rightEvents = std::nullopt;
        }

        // Wait for a small amount of time to avoid CPU overhaul
        cv::waitKey(1);
    }

    return 0;
}
```

```
import dv_processing as dv
import cv2 as cv
from datetime import timedelta

# Path to a stereo calibration file, replace with a file path on your local file␣
↪system
calibration_file_path = "path/to/calibration.json"

# Load the calibration file
calibration = dv.camera.CalibrationSet.LoadFromFile(calibration_file_path)

# It is expected that calibration file will have "C0" as the leftEventBuffer camera
left_camera = calibration.getCameraCalibration("C0")

# The second camera is assumed to be rightEventBuffer-side camera
right_camera = calibration.getCameraCalibration("C1")

# Open the stereo camera with camera names from calibration
capture = dv.io.StereoCapture(left_camera.name, right_camera.name)

# Make sure both cameras support event stream output, throw an error otherwise
if not capture.left.isEventStreamAvailable() or not capture.right.
↪isEventStreamAvailable():
    raise RuntimeError("Input camera does not provide an event stream.")

# Matching window size for the block matcher
window = (24, 24)

# Minimum disparity value to measure
min_disparity = 0

# Maximum disparity value
```

(continues on next page)

(continued from previous page)

```
max_disparity = 40

# Minimum z-score value that a valid match can have
min_score = 0.0

# Initialize the block matcher with rectification
matcher = dv.SparseEventBlockMatcher(dv.camera.StereoGeometry(left_camera, right_
↪camera), window, max_disparity,
                                     min_disparity, min_score)

# Initialization of a stereo event sliver
slicer = dv.StereoEventStreamSlicer()

# Initialize a window to show previews of the output
cv.namedWindow("Preview", cv.WINDOW_NORMAL)

# Local event buffers to implement overlapping window of events for accumulation
global left_event_buffer, right_event_buffer
left_event_buffer = dv.EventStore()
right_event_buffer = dv.EventStore()

# Use one third of the resolution as count of events per accumulated frame
event_count = int((left_camera.resolution[0] * left_camera.resolution[1]) / 3)


# Stereo slicer callback method
def callback(left_events: dv.EventStore, right_events: dv.EventStore):
    # Push input events into the local buffers
    global left_event_buffer, right_event_buffer
    left_event_buffer.add(left_events)
    right_event_buffer.add(right_events)

    # If the number of events is above the count, just keep the latest events
    if len(left_event_buffer) > event_count:
        left_event_buffer = left_event_buffer.sliceBack(event_count)
    if len(right_event_buffer) > event_count:
        right_event_buffer = right_event_buffer.sliceBack(event_count)

    # Number of clusters to extract
    num_clusters = 100

    # Initialize the mean-shift clustering algorithm
    mean_shift = dv.cluster.mean_shift.MeanShiftEventStoreAdaptor(left_event_buffer,␣
↪10, 1, 20, num_clusters)

    # Find cluster centers which are going to be used for disparity estimation
    centers = mean_shift.findClusterCentresEpanechnikov()

    # Run disparity estimation, the output will contain a disparity estimate for each␣
↪of the given points.
    estimates = matcher.computeDisparitySparse(left_event_buffer, right_event_buffer,␣
↪list(map(lambda x: x.pt,
                                                                                    ␣
↪          centers)))

    # Convert the accumulated frames into colored images for preview.
    images = []
```

(continues on next page)

(continued from previous page)

```
    images.append(cv.cvtColor(matcher.getLeftFrame().image, cv.COLOR_GRAY2BGR))
    images.append(cv.cvtColor(matcher.getRightFrame().image, cv.COLOR_GRAY2BGR))

    # Visualize the matched blocks
    index = 0
    for point in estimates:
        # If point estimation is invalid, do not show a preview of it
        if not point.valid:
            continue

        # The rest of the code performs drawing of the match according to the disparity value on the
        # preview images.
        color = dv.visualization.colors.someNeonColor(index)
        index += 1

        # Draw some nice colored markers and rectangles.
        cv.drawMarker(images[1], point.matchedPosition, color, cv.MARKER_CROSS, 7)
        cv.rectangle(images[1],
                     (int(point.matchedPosition[0] - (window[0] / 2)), int(point.matchedPosition[1] - (window[1] / 2))),
                     (int(point.matchedPosition[0] + (window[0] / 2)), int(point.matchedPosition[1] + (window[1] / 2))),
                     color)
        cv.rectangle(
            images[0],
            (int(point.templatePosition[0] - (window[0] / 2)), int(point.templatePosition[1] - (window[1] / 2))),
            (int(point.templatePosition[0] + (window[0] / 2)), int(point.templatePosition[1] + (window[1] / 2))), color)

    # Concatenate images and show them in a window
    cv.imshow("Preview", cv.hconcat(images))


# Register a callback to be done at 30Hz
slicer.doEveryTimeInterval(timedelta(milliseconds=33), callback)

# Buffer input events in these variables to synchronize inputs
left_events = None
right_events = None

# Run the processing loop while both cameras are connected
while capture.left.isRunning() and capture.right.isRunning():
    # Read events from respective left / right cameras
    if left_events is None:
        left_events = capture.left.getNextEventBatch()
    if right_events is None:
        right_events = capture.right.getNextEventBatch()

    # Feed the data into the slicer and reset the buffer
    if left_events is not None and right_events is not None:
        slicer.accept(left_events, right_events)
        left_events = None
        right_events = None

    #  Wait for a small amount of time to avoid CPU overhaul
```

(continues on next page)

(continued from previous page)

```
135     cv.waitKey(1)
```

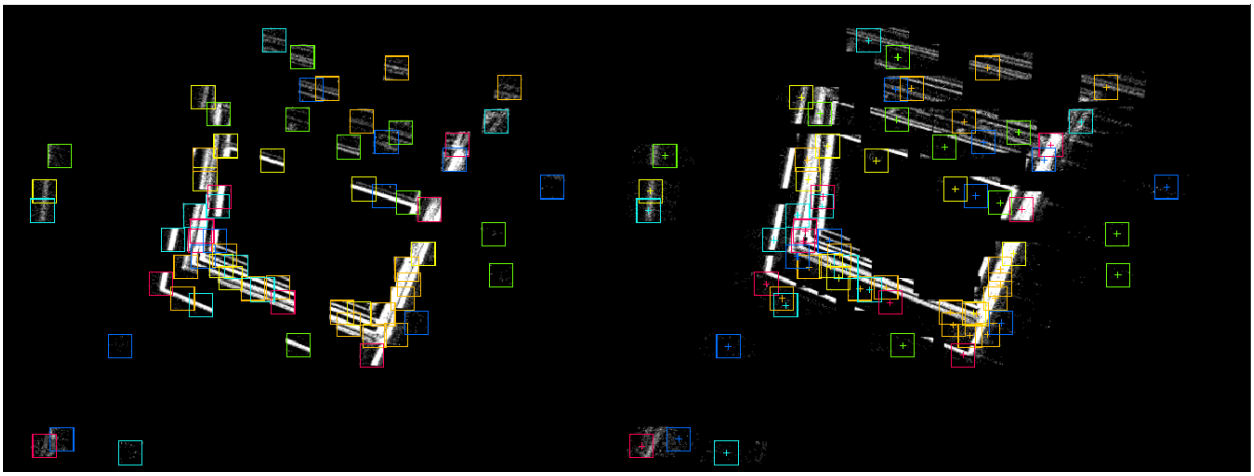


Fig. 2: Expected result of sparse disparity estimation. The colored rectangles represent sparse blocks that are matched on the right side image. Block colors are matched on both images. Note that frame are sparse as well - the accumulation happens only in relevant areas around points of interest. The points of interest are selected on high density event areas as per mean-shift cluster extraction.

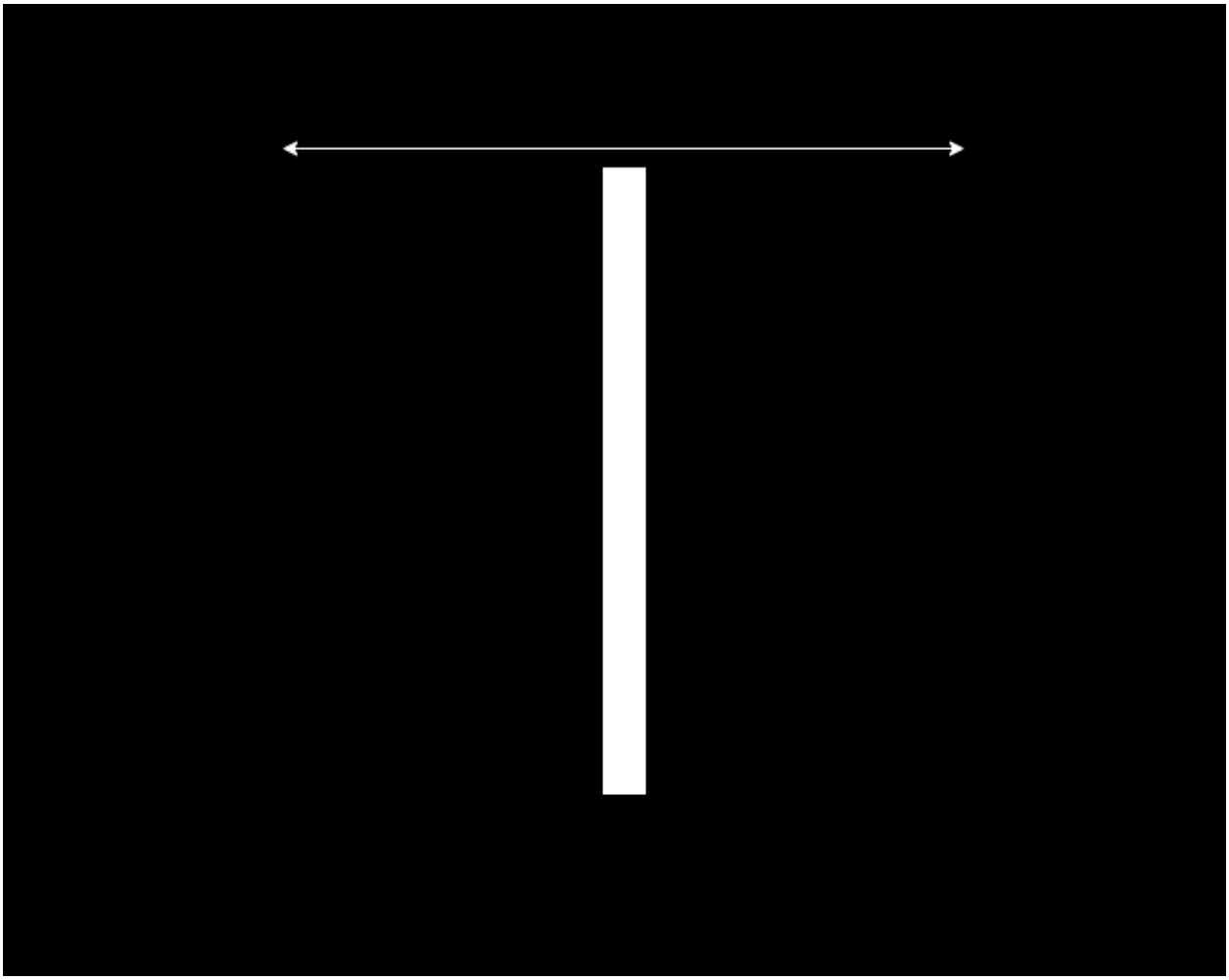
5.2 Motion Compensation

5.2.1 What is the motion compensation algorithm?

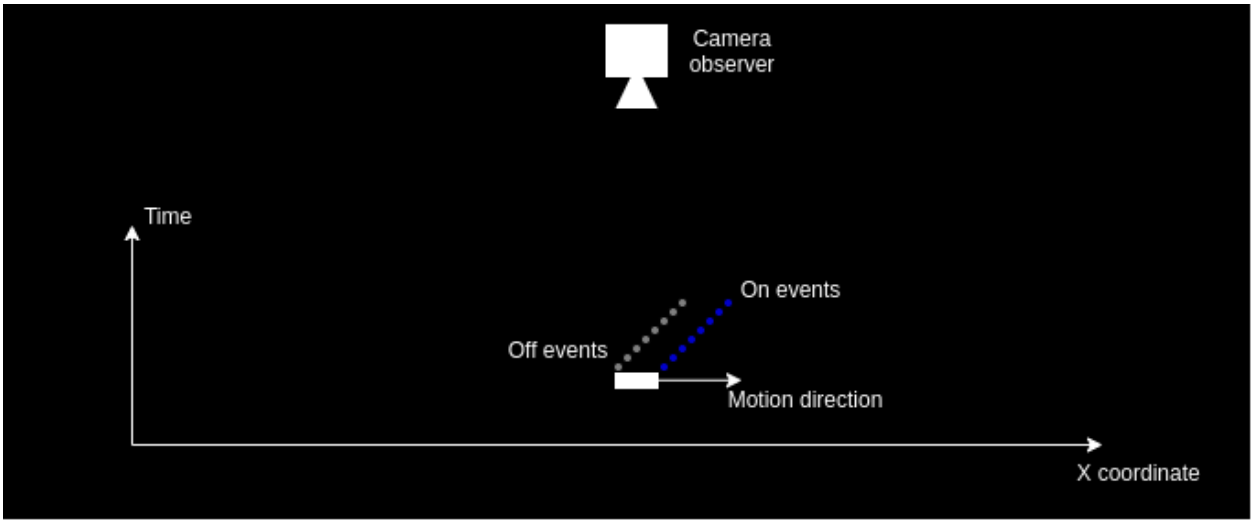
An algorithm to align events in a stream to produce sharp images of edges. Due to camera motion, the events are distributed within the pixel space of a moving camera sensor. If the camera geometry (calibration) and camera motion is known, the camera motion can be applied to the incoming events, so the event locations align to same pixel locations over time.

5.2.2 Visual explanation

Let's discuss a scenario where a white rectangle is moving at a constant speed on a white background. Let's say it's only moving back-and-forth within the X dimension (image below):



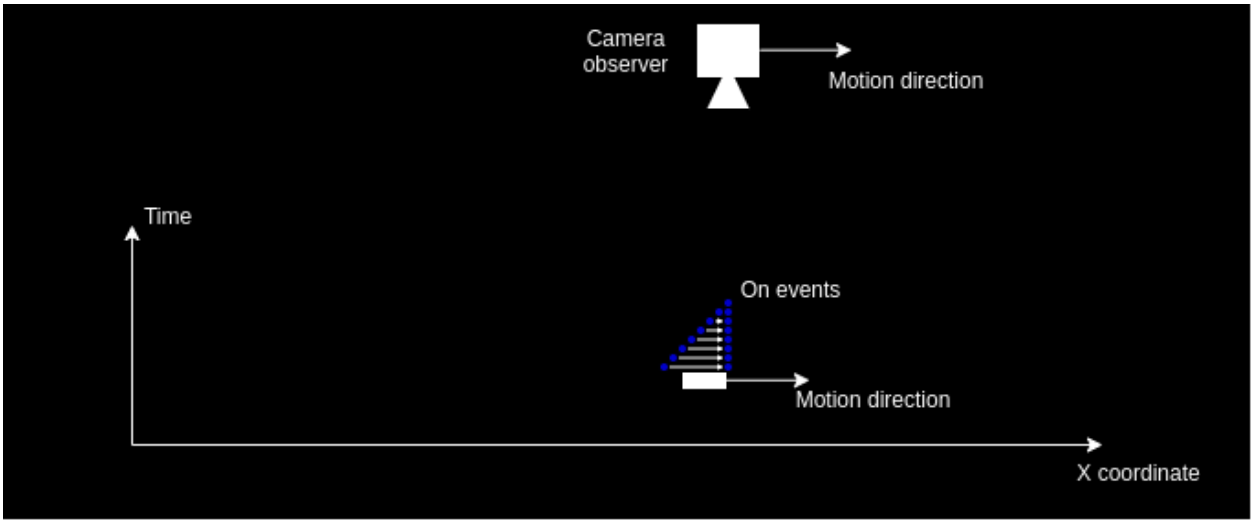
The motion of the rectangle will generate events over time. In the direction of motion the rectangle will generate ON events at the front of rectangle and OFF events at the opposite side. Let's ignore the Y axis and let's look at the event stream from Y axis perspective (given Z axis is time) at the image below.



So the events come in at a slope, so if we accumulate the slice of events they will be represented as a thicker band of ON and OFF events instead of seeing the corresponding edge. This happens due to the fact that the motion happened over

time and event the motion within time-slice is accumulated.

A way to correct this would be to move the camera "virtually" in same direction and speed. This would cause the events to align and be represented at the same pixel location on the sensor (see image below, for simplicity it shows only the ON events aligned).



This is just a scenario to imagine on the concept itself. Usual application case would be when the environment of the camera is static and only the camera moves. If we know the camera motion, we can apply the known motion to the received event coordinate to match the camera motion. In that case the algorithm reverses the problem, assumes the camera is stationary at a single point in time and aligns measured events to happen at the same pixel location in the sensor at a given perspective.

5.2.3 Details of the implementation

Motion compensation applies measured motion of the camera to the event stream. The approach doesn't cover how the motion is estimated: it can be either some external sensor (e.g. IMU) or visual odometry system estimating camera ego-motion, these measurements are assumed to be known by the algorithm.

Camera geometry

To apply known motion of the camera, firstly event pixel coordinates must be represented in a 3D space. This is performed using camera geometry and back-projection. Back-projection is performed to find a pixel coordinate ray that maps pixel coordinates into a flat 3D plane in front of the camera, that is one-meter away from the focal point.

Camera matrix (K in the formula below) contains focal length (fx, fy) and principal point (cx, cy) coordinates retrieved from calibration of the camera sensor.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Back-projection can be achieved by multiplying pixel coordinate homogenous representation of [px, py, 1.0] multiplied (px and py are the pixel coordinates on the sensor) by the inverse of the camera matrix.

$$R = K^{-1} \cdot \begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix}$$

This estimated ray (R) can be multiplied by an estimated depth distance (d) to the scene provide a reasonable estimate of the actual 3D position that cause the pixel to generate an event (let's call it P). If the distance to the scene is not known, a guess of 3 meters is usually a good enough estimate.

$$P = R \cdot d$$

Kinematics (applying the motion)

Let's denote a transformation matrix that describes a motion in 3D, 4x4 matrix containing a 3x3 rotation matrix and 3x1 translation vector.

$$T = \begin{bmatrix} r & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{00} & r_{01} & r_{02} & t_x \\ r_{10} & r_{11} & r_{12} & t_y \\ r_{20} & r_{21} & r_{22} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

By multiplying a 3D point P (that was captured at time t1) by the matrix that describes camera motion between time-points t1 and t2, we can estimate the position of the object at camera perspective t2.

$$\begin{bmatrix} P_{t_2} \\ 1 \end{bmatrix} = T_{t_1}^{t_2} \cdot \begin{bmatrix} P_{t_1} \\ 1 \end{bmatrix}$$

The resulting point at t2 can be normalized by the Z axis (X, Y, and Z components divided by Z) results in a projection at one meter plane.

$$R_{t_2} = \hat{P}_{t_2}$$

which can be projected into pixel frame by multiplying the point by camera matrix K, thus aligning the pixel coordinates in space.

$$\begin{bmatrix} p_x^{t_2} \\ p_y^{t_2} \\ 1 \end{bmatrix} = K \cdot R_{t_2}$$

5.2.4 Usage of available implementation

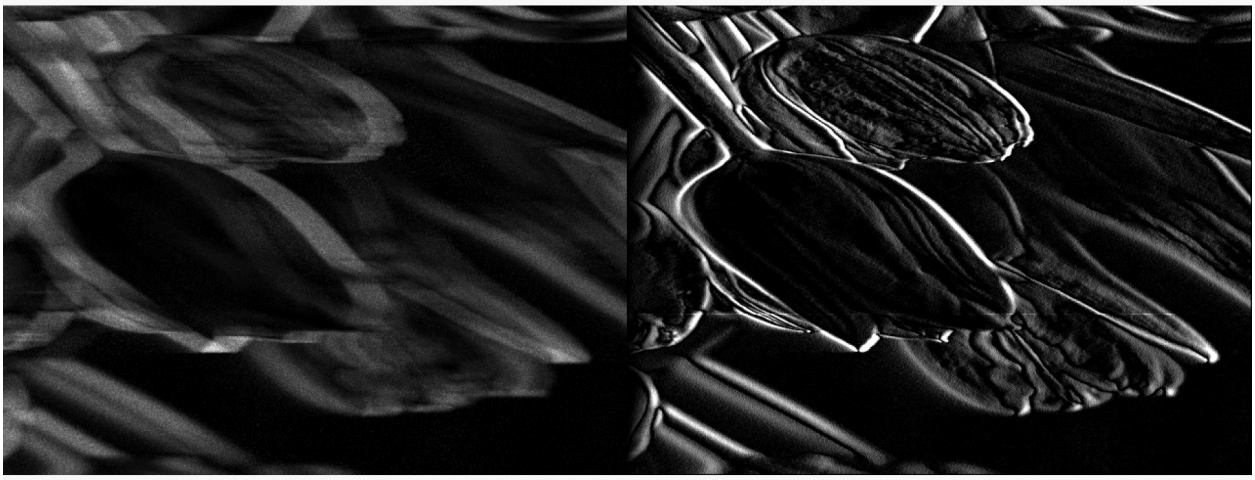
The motion compensation algorithm is available in C++ and Python.

A sample usage of the motion compensator can be found in `samples/motion-compensation-samples/imu_compensation.cpp`. In this sample, measurement from a gyroscope are concatenated to estimate camera rotational motion, this information is then used to reduce motion blur induced by the camera motion.

5.3 Contrast Maximization

5.3.1 What is contrast maximization?

Given a set of events obtained from the camera and/or scene motion, contrast maximization seeks to find the optimal parameters of the motion underlying the generated events. The movement is modeled a-priori (shared by all events) and defines the set of parameters to be optimized. Those parameters are optimized using non-linear optimization, which aims at maximizing the contrast of the event image generated by warped events.



The image on the top shows a sample of an accumulated event image without motion compensation (left) versus an event image after warping events using contrast maximization. Note that, assuming no brightness variation in the scene, each event carries camera or scene motion information.

Examples of possible motions to be optimized are:

- Camera translational motion
- Camera rotational motion
- Full camera ego-motion (translation + depth)
- Optical flow
- Scene depth

Let's now explain the idea of contrast maximization more in detail for pure camera motion (i.e. static scene). The same principles can be applied to a dynamic scene without loss of generality.

Algorithm in details

Events will be generated by a static scene and a camera moving in space due to the camera motion. All events corresponding to the same 3D point in the scene will fall at the same location in the pixels space. Taking a set of events in a given time interval, if the camera's motion in this interval is perfectly known, it is possible to warp events in space to a given previous point in time by applying the inverse of the camera motion. On the other hand, if events are warped with incorrect motion, they will fall at different locations. Based on this simple idea, it is possible to build an objective function that estimates the motion estimate's goodness by analyzing warped events. This is done by measuring the sharpness of the event image generated by warping events with the assumed camera motion.

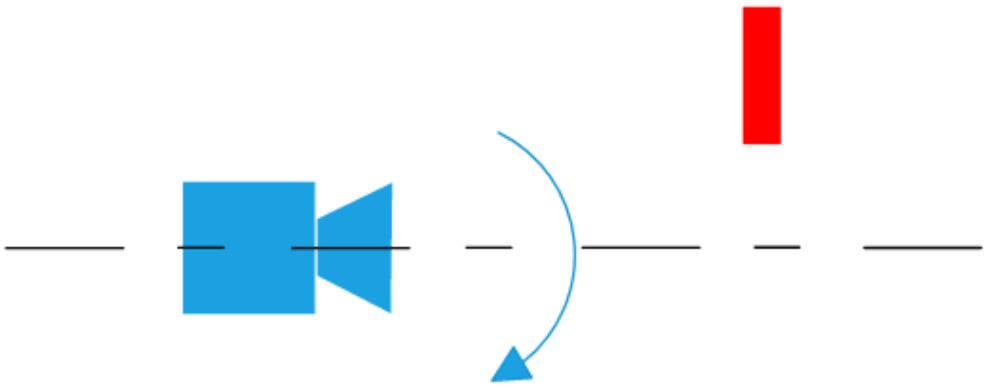
What is contrast?

Since the algorithm's name is contrast maximization, it is important to understand where the name comes from. Indeed, the contrast is the value of the cost function used in the optimization. The contrast of the event image corresponds to computing its variance. If the events are warped with the wrong motion assumption, they will fall at random locations on the image plane. Assuming enough events, there will be at least one event per pixel in the image plane in the extreme case. In this case, the contrast of the image will be 0. On the other hand, if an event image has strong edges, the contrast will be high. Because event-cameras detect edges, it is reasonable to find the camera motion by warping events such that those edges are "reconstructed" at a specific time.

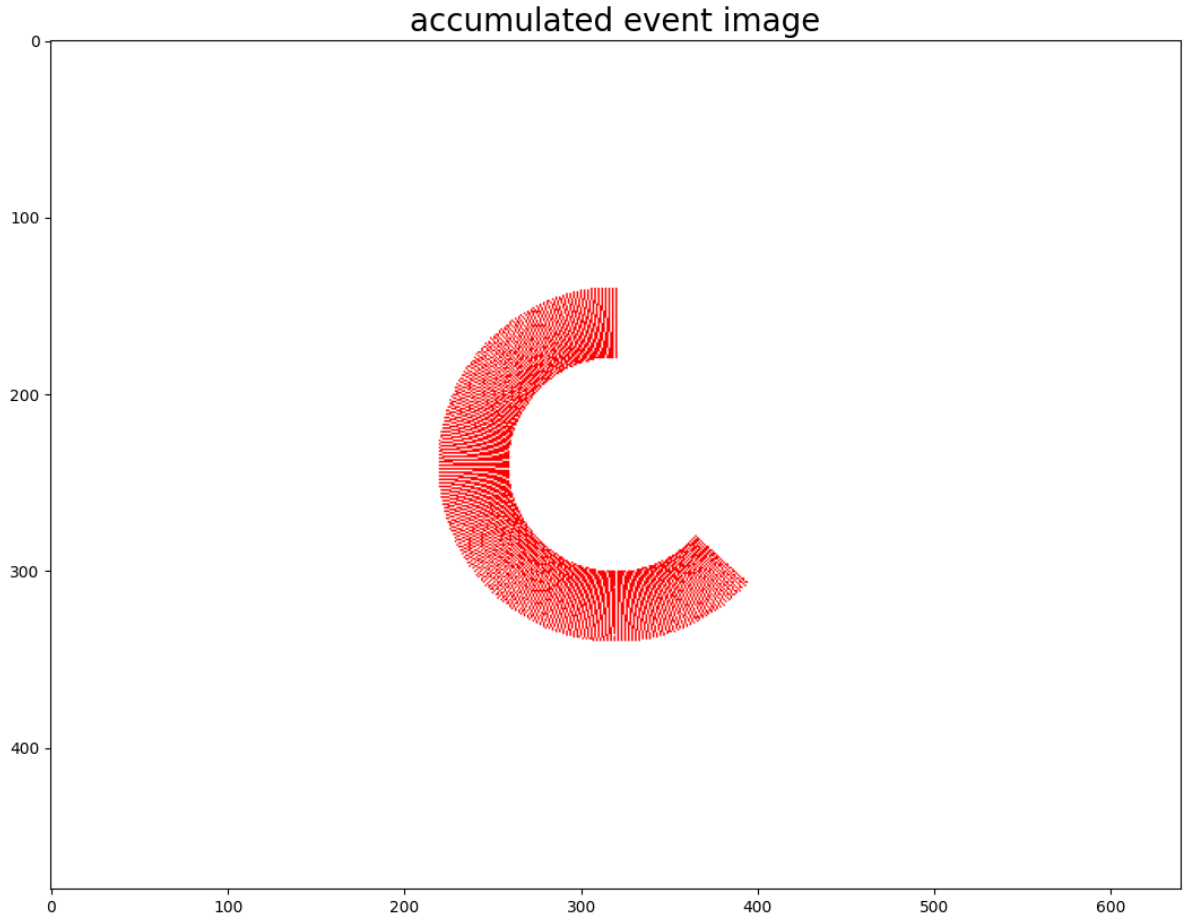
To understand this principle, below you can find an example of how contrast maximization would work far a rotating camera in front of a static bar.

5.3.2 Example: static bar - rotating camera

To understand the basic principle of contrast maximization, we use a simple scene composed of a static bar and a camera pointing to it. We assume the camera motion is defined by a pure rotation around its optical axis. As long as the camera is not moving, no event is generated. The image below shows the example setup viewed from the side.



An x-axis point defines the camera coordinate system to the right, the y-axis to the bottom and the z-axis pointing in front. Once the camera starts rotating at constant speed *ωcamera* (this value is not known) around its optical axis, events due to the relative motion between the static bar and the camera will be generated on the image plane. By reconstructing the event image corresponding to the accumulated events, we would get an image similar to the one on the bottom.



Now we explain how to estimate the camera speed *ωcamera* from the event set *Et0-t1* generated in the time interval between *t0* and *t1*. Select motion model used to explain generated events. We assume the camera to rotate only. The camera motion is defined by a pure rotation $\omega k = [\omega x, * \omega y^*, \omega z]$. Since the camera rotate only around its optical axis we set $\omega x = 0$, $\omega y = 0$ and optimize only ωx, ωz value.

Here is a pseudocode of the non-linear optimization to be solved:

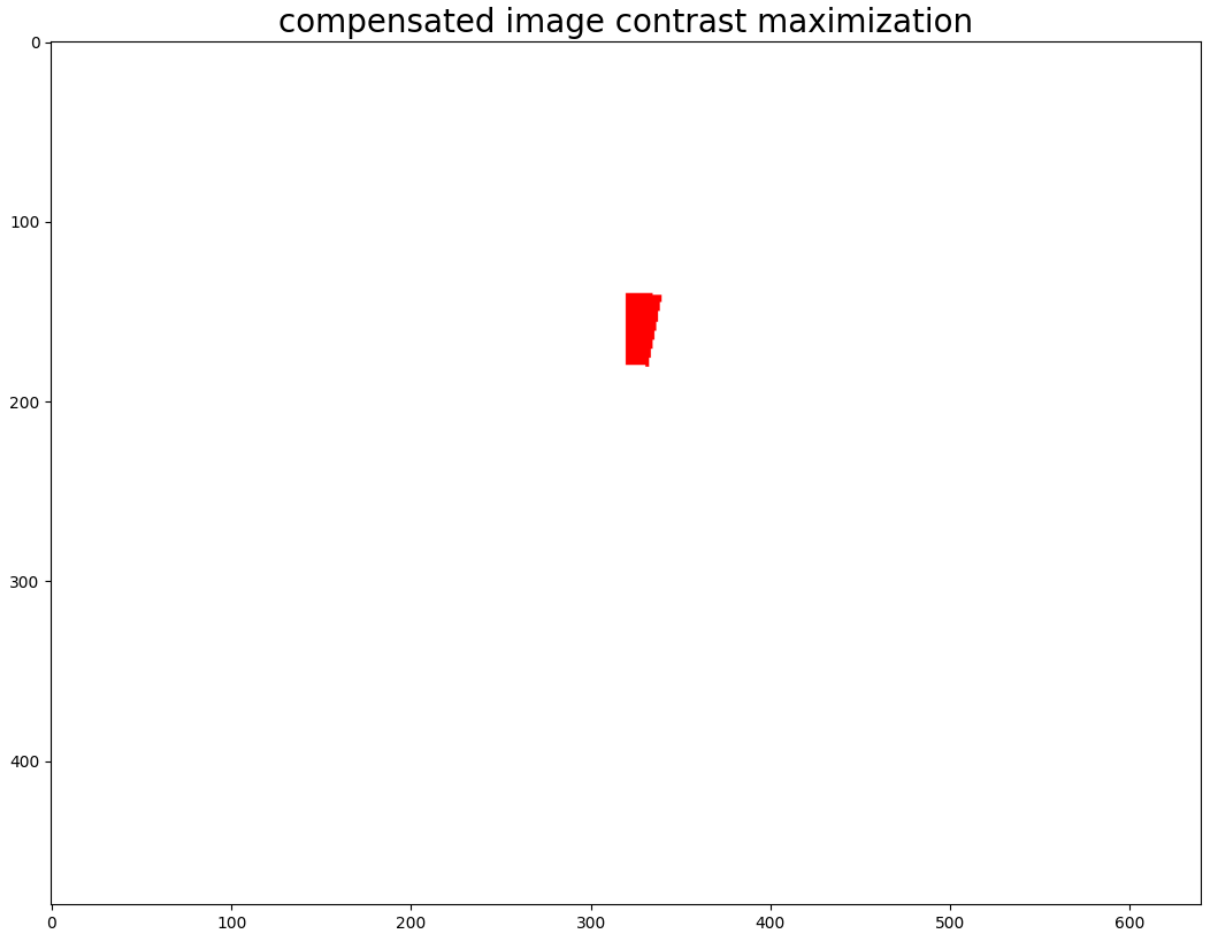
- Solve for *ωopt* by maximizing variance.
- While (not_converged):
  - current camera speed guess: ωk
  - for each event *ei== [xi, yi, poli, ti]*:
    - compute the total camera motion between t0 and ti as: $\theta z = \omega z(k)$
      - (t i - t0). Since the camera rotate only along z axis, it implies $\theta = [0, 0, \theta z]$
    - get equivalent camera rotation matrix R defined by θ.

$$p_{t(0)} = K^{-1}R^{-1}p_{t(i)}$$

    - warp event *ei* to time t0 using where K is che camera intrinsic matrix, R is camera rotation matrix defined by θ and pt(i) = [xi, yi, 1].
  - Get set of warped events Ewarped, representing all events warped at time t0.
  - Compute event image from Ewarped.
  - Compute variance of event image

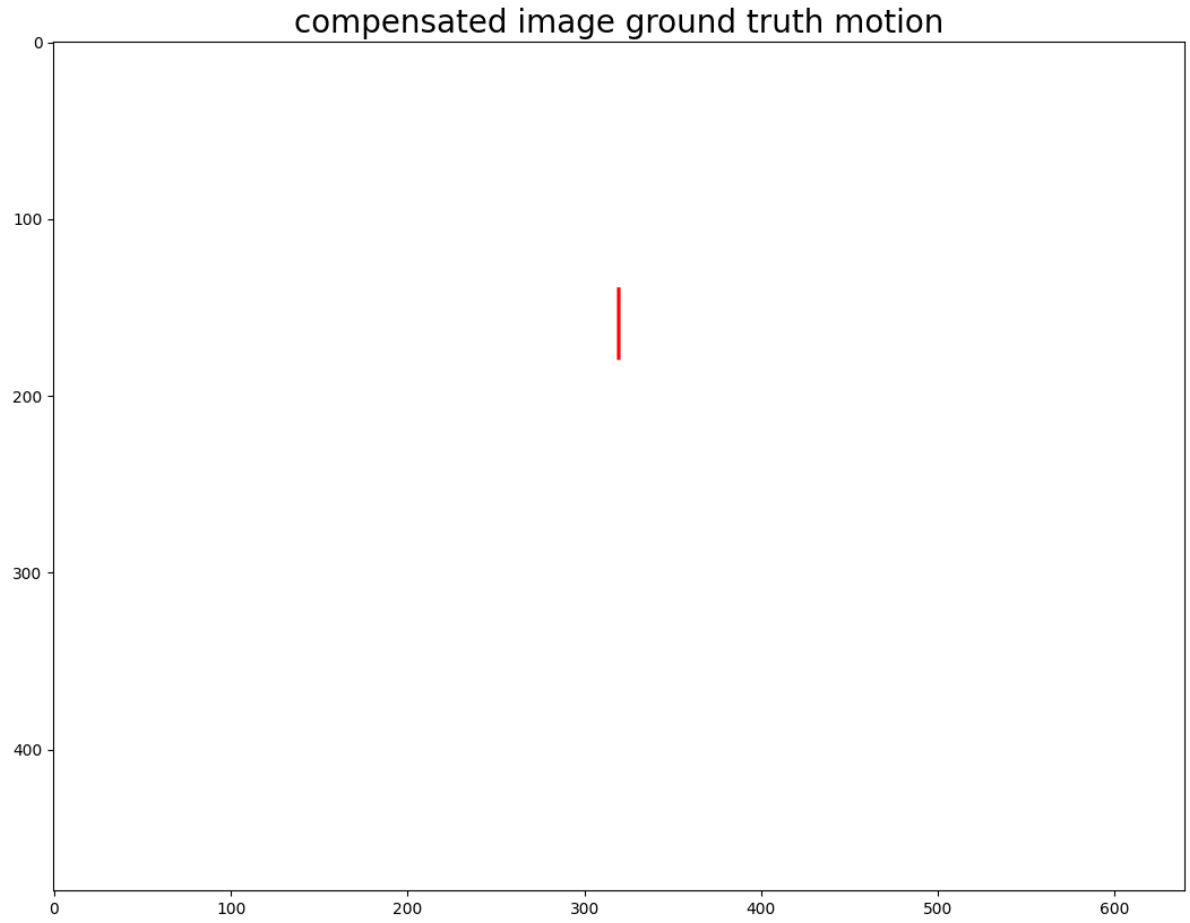
Usually, the camera motion is defined by a 4x4 transformation. In this case, since a pure rotational model is assumed, we neglect the translational part. Because of this, the whole camera motion can be defined by a 3x3 matrix representing the camera rotation.

At each iteration, if the value of ωk is close to $\omega camera$ events will be warped in such a way that the event image will be similar to the image below. Vice versa, if the value of ωk is far from ω *camera*, then the events will not fall along the vertical bar, and the generated event image will have lower contrast. The non-linear optimization computes the gradient by evaluating multiple event images at time *k*, and this gradient is then used to update the camera speed for the next iteration. If the algorithm converges correctly (i.e. ωopt~ωcamera), the resulting image will be similar to the one shown on the bottom.



Note that the algorithm rarely converges to the exact motion underlying the generated events. In most cases, the results will be very close to the ground truth motion (if the initialization of the parameters to be optimized is good enough). For completeness, the image at the bottom shows the compensated image assuming perfect motion from the camera is known.

As you can see, all events converge to the same vertical line when motion is compensated with ground truth motion.



Summary: algorithm steps

In summary, the steps of the contrast maximization algorithm for a given motion model are the following:

1. Warp events according to the trajectories defined by the model and its model parameters.
2. Generate event image from warped events.
3. Compute score based on the image of warped events.
4. Optimize score for model parameters.

Importance of initial guess

Since contrast maximization is based on a non-linear optimization, the initial guess for the parameters to be optimized has a substantial impact on the accuracy of the final optimization. If the initial guess is very different from the proper motion, the non-linear optimization will most likely end in a local minimum or converge to unexpected results.

5.3.3 Usage of available implementation

The contrast maximization algorithm is available in C++.

There are two sample usages. The first one can be found at `samples/motion-compensation-samples/contrast_maximization_rotation.cpp`: the gyroscope offsets along x, y, and z are optimized in this sample. The second one can be found at `samples/motion-compensation-samples/contrast_maximization_translation_depth.cpp`: here, the camera translation alone on the three-axis is optimized, together with scene depth.

It is possible to run the code `samples/motion-compensation-samples/contrast_maximization_translation_depth.cpp` using a sample recording that can be downloaded from https://s3.eu-central-1.amazonaws.com/release.inivation.com/datasets/translation_tunnel_sideways.aedat4. The code can be run by passing as argument the path to the downloaded aedat4 file, the path of the corresponding calibration file that can be found in /docs/source/assets/contrast_maximization/calibration.json and as initial translation and depth guesses:

depth = 2, *translation = [-0.12, 0.0, 0.0]*.

More information about the two samples can be found in the code.

5.3.4 Limitations

This algorithm is very powerful, but at the same time, it has some inherent limitations that need to be taken into account.

Wrong motion model

The results would not be very accurate if the motion model assumed differs from the underlying motion that generates the event set. This can be the case if multiple motions are present in the scene (e.g., camera motion + dynamic scene). It would be necessary to first cluster events into separate groups based on similar motion and then apply contrast maximization on each event set separately.

Too complex optimization

Another possible cause that might lead to incorrect results is the case in which the optimization is too complex. For example, optimizing both camera rotation and translation simultaneously will create a vast search space for the optimization. The optimization result might degenerate into an unexpected output that does not consider the scene's geometry.

Events not generated by motion

Contrast maximization assumes all events generated by motion, either from the camera or from the scene. However, suppose events are generated from brightness change. Contrast maximization will try to explain those events with some motion, even if events are caused by brightness change and not movement. Due to this, the optimization will converge to unexpected results.

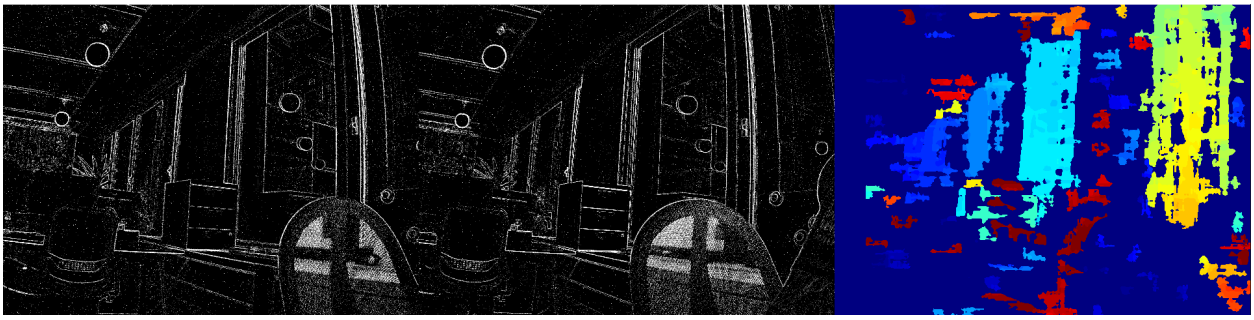


Fig. 3: Expected result of semi-dense disparity estimation. The output provides two accumulated frames and color-coded disparity map.

CHAPTER

SIX

API

API documentation provides references for available classes and methods in the library. You can find detailed documentation on each method for their use case, function arguments, and produced outputs.

6.1 API

Full API documentation, automatically generated from doxygen comments.

class **Accumulator** : public *dv*::*AccumulatorBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/frame/accumulator.hpp> Common accumulator class that allows to accumulate events into a frame. The class is highly configurable to adapt to various use cases. This is the preferred functionality for projecting events onto a frame.

Accumulation of the events is performed on a floating point frame, with every event contributing a fixed amount to the potential. Timestamps of the last contributions are stored as well, to allow for a decay.

Due to performance, no check on the event coordinates inside image plane is performed, unless compiled specifically in DEBUG mode. Events out of the image plane bounds will result in undefined behaviour, or program termination in DEBUG mode.

Public Types

enum class **Decay**

Decay function to be used to decay the surface potential.

- `NONE`: Do not decay at all. The potential can be reset manually by calling the `clear` function
- `LINEAR`: Perform a linear decay with given slope. The linear decay goes from currentpotential until the potential reaches the neutral potential
- `EXPONENTIAL`: Exponential decay with time factor tau. The potential eventually converges to zero.
- `STEP`: Decay sharply to neutral potential after the given time. Constant potential before.

Values:

enumerator **NONE**

enumerator **LINEAR**

enumerator **EXPONENTIAL**

enumerator **STEP**

Public Functions

inline **Accumulator**()

Silly default constructor. This generates an accumulator with zero size. An accumulator with zero size does not work. This constructor just exists to make it possible to default initialize an *Accumulator* to later redefine.

inline explicit **Accumulator**(const cv::Size &resolution, *Accumulator*::Decay decayFunction = *Decay*::*EXPONENTIAL*, double decayParam = 1.0e+6, bool synchronousDecay = false, float eventContribution = 0.15f, float maxPotential = 1.0f, float neutralPotential = 0.f, float minPotential = 0.f, bool ignorePolarity = false)

Accumulator constructor Creates a new *Accumulator* with the given params. By selecting the params the right way, the *Accumulator* can be used for a multitude of applications. The class also provides static factory functions that adjust the parameters for common use cases.

Parameters

- **resolution** – The size of the resulting frame. This must be at least the dimensions of the eventstream supposed to be added to the accumulator, otherwise this will result in memory errors.
- **decayFunction** – The decay function to be used in this accumulator. The decay function is one of `NONE`, `LINEAR`, `EXPONENTIAL`, `STEP`. The function behave like their mathematical definitions, with LINEAR AND STEP going back to the `neutralPotential` over time, EXPONENTIAL going back to 0.
- **decayParam** – The parameter to tune the decay function. The parameter has a different meaning depending on the decay function chosen: `NONE`: The parameter is ignored `LINEAR`: The paramaeter describes the (negative) slope of the linear function `EXPONENTIAL`: The parameter describes tau, by which the time difference is divided.
- **synchronousDecay** – if set to true, all pixel values get decayed to the same time as soon as the frame is generated. If set to false, pixel values remain at the state they had when the last contribution came in.
- **eventContribution** – The contribution a single event has onto the potential surface. This value gets interpreted positively or negatively depending on the event polarity
- **maxPotential** – The upper cut-off value at which the potential surface is clipped
- **neutralPotential** – The potential the decay function converges to over time.
- **minPotential** – The lower cut-off value at which the potential surface is clipped
- **ignorePolarity** – Describes if the polarity of the events should be kept or ignored. If set to true, all events behave like positive events.

inline virtual void **accumulate**(const *EventStore* &packet) override

Accumulates all the events in the supplied packet and puts them onto the accumulation surface.

Parameters

packet – The packet containing the events that should be accumulated.

inline virtual *dv*::*Frame* **generateFrame**() override

Generates the accumulation frame (potential surface) at the time of the last consumed event. The function writes the output image into the given `frame` argument. The output frame will contain data with type CV_8U.

Parameters
frame – the frame to copy the data to

inline void **clear**()

Clears the potential surface by setting it to the neutral value. This function does not reset the time surface.

inline void **setRectifyPolarity**(bool rectifyPolarity)

If set to true, all events will incur a positive contribution to the potential surface

Deprecated:

Use *setIgnorePolarity()* method instead.

See also:

dv::Accumulator::setIgnorePolarity

Parameters
rectifyPolarity – The new value to set

inline void **setIgnorePolarity**(const bool ignorePolarity)

If set to true, all events will incur a positive contribution.

Parameters
ignorePolarity – The new value to set

inline void **setEventContribution**(float eventContribution)

Contribution to the potential surface an event shall incur. This contribution is either counted positively (for positive events or when `rectifyPolatity` is set).

Parameters
eventContribution – The contribution a single event shall incur

inline void **setMaxPotential**(float maxPotential)

Parameters
maxPotential – the max potential at which the surface should be capped at

inline void **setNeutralPotential**(float neutralPotential)

Set a new neutral potential value. This will also reset the cached potential surface to the given new value.

Parameters
neutralPotential – The neutral potential to which the decay function should go. Exponential decay always goes to 0. The parameter is ignored there.

inline void **setMinPotential**(float minPotential)

Parameters
minPotential – the min potential at which the surface should be capped at

inline void **setDecayFunction**(*Decay* decayFunction)

Parameters
decayFunction – The decay function the module should use to perform the decay

inline void **setDecayParam**(double decayParam)

The decay param. This is slope for linear decay, tau for exponential decay

Parameters
decayParam – The param to be used

inline void **setSynchronousDecay**(bool synchronousDecay)

If set to true, all valued get decayed to the frame generation time at frame generation. If set to false, the values only get decayed on activity.

Parameters
synchronousDecay – the new value for synchronoues decay

inline bool **isRectifyPolarity**() const

Check whether polarity rectification (ignorePolarity) is enabled.

Deprecated:

Use *isIgnorePolarity()* method instead.

See also:

dv::Accumulator::isIgnorePolarity

Returns
True if enabled, false otherwise.

inline bool **isIgnorePolarity**() const

Check whether polarity of events is ignored.

Returns
True if polarity is ignored, false otherwise.

inline float **getEventContribution**() const

inline float **getMaxPotential**() const

inline float **getNeutralPotential**() const

inline float **getMinPotential**() const

inline *Decay* **getDecayFunction**() const

inline double **getDecayParam**() const

inline *Accumulator* &**operator<<**(const *EventStore* &store)

Accumulates the event store into the accumulator.

Parameters
store – The event store to be accumulated.

Returns
A reference to this *Accumulator*.

inline cv::Mat **getPotentialSurface**() const

Retrieve a copy of the currently accumulated potential surface. Potential surface contains raw floating point values aggregated by the accumulator, the values are within the configured range of [minPotential; maxPotential]. This returns a deep copy of the potential surface.

Returns
Potential surface image containing CV_32FC1 data.

Private Functions

inline void **decay**(int16_t x, int16_t y, int64_t time)

INTERNAL_USE_ONLY Decays the potential at coordinates x, y to the given time, respecting the decay function. Updates the time surface to the last decay.

Parameters

- **x** – The x coordinate of the value to be decayed
- **y** – The y coordinate of the value to be decayed
- **time** – The time to which the value should be decayed to.

inline void **contribute**(int16_t x, int16_t y, bool polarity)

INTERNAL_USE_ONLY Contributes the effect of a single event onto the potential surface.

Parameters

- **x** – The x coordinate of where to contribute to
- **y** – The y coordinate of where to contribute to
- **polarity** – The polarity of the contribution

Private Members

bool **rectifyPolarity_** = false

float **eventContribution_** = .0

float **maxPotential_** = .0

float **neutralPotential_** = .0

float **minPotential_** = .0

Decay **decayFunction_** = *Decay*::*NONE*

double **decayParam_** = .0

bool **synchronousDecay_** = false

TimeSurface **decayTimeSurface_**

cv::Mat **potentialSurface_**

int64_t **highestTime_** = 0

int64_t **lowestTime_** = -1

bool **resetTimestamp** = true

class **AccumulatorBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/frame/accumulator_base.hpp> An accumulator base that can be used to implement different types of accumulators. Two provided implementations are the `dv::Accumulator` which is highly configurable and provides numerous ways of generating a frame from events. Another implementation is the `dv::EdgeMapAccumulator` which accumulates event in a histogram representation with configurable contribution, but it is more efficient compared to generic accumulator since it uses 8-bit unsigned integers as internal memory type.

Subclassed by *dv::Accumulator*, *dv::EdgeMapAccumulator*

Public Types

typedef *std*::shared_ptr<*AccumulatorBase*> **SharedPtr**

typedef *std*::unique_ptr<*AccumulatorBase*> **UniquePtr**

Public Functions

inline explicit **AccumulatorBase**(const cv::Size &shape)

Accumulator constructor from known event camera sensor dimensions.

Parameters
shape – Sensor dimensions

virtual void **accumulate**(const *EventStore* &packet) = 0

Accumulate given event store packet into a frame.

Parameters
packet – Event packet to be accumulated.

inline const cv::Size &**getShape**() const

Get the image dimensions expected by the accumulator.

Returns
Image dimensions

virtual *dv*::*Frame* **generateFrame**() = 0

Generates the accumulation frame (potential surface) at the time of the last consumed event. The function returns an OpenCV frame to work with.

Returns
An OpenCV frame containing the accumulated potential surface.

inline *dv*::*Frame* &**operator>>**(*dv*::*Frame* &mat)

Output stream operator support for frame generation.

Parameters
mat – Output image

Returns
Output image

inline void **accept**(const *EventStore* &packet)

Accumulate the given packet.

Parameters
packet – Input event packet.

virtual **~AccumulatorBase**() = default

Protected Attributes

cv::Size **shape_**

template<*concepts*::*AddressableEvent* **EventType**, class **EventPacketType**>

class **AddressableEventStorage**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> EventStore class. An `EventStore` is a collection of consecutive events, all monotonically increasing in time. EventStore is the basic data structure for handling event data. Event packets hold their data in shards of fixed size. Copying an `EventStore` results in a shallow copy with shared ownership of the shards that are common to both EventStores. EventStores can be sliced by number of events or by time. Slicing creates a shallow copy of the `EventPackage`.

Public Types

using **value_type** = *EventType*

using **const_value_type** = const *EventType*

using **pointer** = *EventType**

using **const_pointer** = const *EventType**

using **reference** = *EventType*&

using **const_reference** = const *EventType*&

using **size_type** = size_t

using **difference_type** = ptrdiff_t

using **packet_type** = *EventPacketType*

using **const_packet_type** = const *EventPacketType*

using **iterator** = *AddressableEventStorageIterator*<*EventType*, *EventPacketType*>

using **const_iterator** = *iterator*

Public Functions

AddressableEventStorage() = default

Default constructor. Creates an empty `EventStore`. This does not allocate any memory as long as there is no data.

inline void **add**(const *AddressableEventStorage* &store)

Merges the contents of the supplied Event Store into the current event store. This operation can cause event data copies if that results in more optimal memory layout, otherwise the operation only performs shallow copies of the data by sharing the ownership with previous event storage. The two event stores have to be in ascending order.

Parameters
store – the store to be added to this store

inline Eigen::Matrix<int64_t, Eigen::Dynamic, 1> **timestamps**() const

Retrieve timestamps of events into a one-dimensional eigen matrix. This performs a copy of the values. The values are guaranteed to be monotonically increasing.

Returns
A one-dimensional eigen matrix containing timestamps of events.

inline Eigen::Matrix<int16_t, Eigen::Dynamic, 2> **coordinates**() const

Retrieve coordinates of events in a 2xN eigen matrix. Method performs a copy of the values. Coordinates maintain the same order as within the event store. First column is the x coordinate, second column is the y coordinate.

Returns
A two-dimensional eigen matrix containing x and y coordinates of events.

inline Eigen::Matrix<uint8_t, Eigen::Dynamic, 1> **polarities**() const

Retrieve polarities of events in a one-dimensional eigen matrix. Method performs a copy of the values. Polarities maintain the same order as within the event store. Polarities are converted into unsigned 8-bit integer values, where 0 stands for negative polarity event and 1 stands for positive polarity event.

Returns
A one-dimensional eigen matrix containing polarities of events.

inline *EigenEvents* **eigen**() const

Convert the event store into eigen matrices. This function performs a deep copy of the memory.

Returns
Events in represented in eigen matrices.

inline explicit **AddressableEventStorage**(*std*::shared_ptr<const *EventPacketType*> packet)

Creates a new `EventStore` with the data from an *`EventPacket`*. This is a shallow operation. No data is copied. The EventStore gains shared ownership of the supplied data. This constructor also allows the implicit conversion from `dv::InputVectorDataWrapper<`*`dv::EventPacket`*`, dv::Event>` to *`dv::AddressableEventStorage`*`<dv::Event,` *`dv::EventPacket`*`>` Implicit conversion intended.

Parameters
packet – the packet to construct the EventStore from

inline *AddressableEventStorage* &**operator=**(*std*::shared_ptr<const *EventPacketType*> packet)

Assignment operator for packet const-pointer type. Will construct a new EventStore within the variable.

Parameters
packet – A pointer to the event data packet.

Returns

inline void **add**(const *EventType* &event)

Adds a single Event to the EventStore. This will potentially allocate more memory when the currently available shards are exhausted. Any new memory receives exclusive ownership by this packet.

Parameters
event – A reference to the event to be added.

inline void **push_back**(const *EventType* &event)

Adds a single Event to the EventStore. This will potentially allocate more memory when the currently available shards are exhausted. Any new memory receives exclusive ownership by this packet.

Parameters
event – A reference to the event to be added.

inline void **push_back**(*EventType* &&event)

Moves a single Event into the EventStore. This will potentially allocate more memory when the currently available shards are exhausted. Any new memory receives exclusive ownership by this packet.

Parameters
event – A movable reference to the event to be added.

template<class ...**Args**>
inline *EventType* &**emplace_back**(*Args*&&... args)

Construct an event at the end of the storage.

Template Parameters
_constr_args – Argument template

Parameters
_args – Argument values

Returns
Reference to the last newly created element

inline *AddressableEventStorage* **operator+**(const *AddressableEventStorage* &other) const

Returns a new EventStore that is the sum of this event store as well as the supplied event store. This is a const operation that does not modify this event store. The returned event store holds all the data of this store and the other. This is a shallow operation, no event data has to be copied for this.

Parameters
other – The other store to be added

Returns
A new EventStore, containing the events from this and the other store

inline *AddressableEventStorage* **operator+**(const *EventType* &event) const

Returns a new event store that contains the same data as this event store, but with the given event added. This is a shallow operation. No event data has to be copied for this.

Parameters
event – The event to be added to this event store

Returns
A new event store containing the same data as the old event store plus the supplied event

inline void **operator+=**(const *AddressableEventStorage* &other)

Adds all the events of the other event store to this event store.

Parameters
other – The event store to be added

inline void **operator+=**(const *EventType* &event)

Adds the provided event to the end of this event store

Parameters
event – The event to be added

inline *AddressableEventStorage* &**operator<<**(const *EventType* &event)

Adds the given event to the end of this EventStore.

Parameters
event – The event to be added

Returns
A reference to this EventStore.

inline size_t **size**() const noexcept

Returns the total size of the EventStore.

Returns
The total size (in events) of the packet.

inline *AddressableEventStorage* **slice**(const size_t start, const size_t length) const

Returns a new EventStore which is a shallow representation of a slice of this EventStore. The slice is from `start` (number of, events, minimum 0, maximum `getLength()`) and has a length of `length`.

As a slice is a shallow representation, no EventData gets copied by this operation. The resulting EventStore receives shared ownership over the relevant parts of the data. Should the original EventStore get out of scope, memory that is not relevant to the sliced EventStore will get freed.

Parameters

- **start** – The start index of the slice (in number of events)
- **length** – The desired length of the slice (in number of events)

Returns
A new EventStore object which references to the sliced, shared data. No Event data is copied.

inline *AddressableEventStorage*<*EventType*, *EventPacketType*> **slice**(const size_t start) const

Returns a new EventStore which is a shallow representation of a slice of this EventStore. The slice is from `start` (number of, events, minimum 0, maximum `getLength()`) and goes to the end of the EventStore. This method slices off the front of an EventStore.

As a slice is a shallow representation, no EventData gets copied by this operation. The resulting EventStore receives shared ownership over the relevant parts of the data. Should the original EventStore get out of scope, memory that is not relevant to the sliced EventStore will get freed.

Parameters
start – The start index of the slice (in number of events). The slice will be from this index to the end of the packet.

Returns
A new EventStore object which references to the sliced, shared data. No Event data is copied.

inline *AddressableEventStorage* **sliceTime**(const int64_t startTime, const int64_t endTime, size_t &retStart, size_t &retEnd) const

Returns a new EventStore which is a shallow representation of a slice of this EventStore. The slice is from a

specific startTime (in event timestamps, microseconds) to a specific endTime (event timestamps, microseconds). The actual size (in events) of the resulting packet depends on the event rate in the requested time interval. The resulting packet may be empty, if there is no event that happened in the requested interval.

As a slice is a shallow representation, no EventData gets copied by this operation. The resulting EventStore receives shared ownership over the relevant parts of the data. Should the original EventStore get out of scope, memory that is not relevant to the sliced EventStore will get freed.

The sliced output will be in the time range [startTime, endTime), endTime is exclusive.

Parameters

- **startTime** – The start time of the required slice (inclusive)
- **endTime** – The end time of the required time (exclusive)
- **retStart** – parameter that will get set to the actual index (in number of events) at which the start of the slice occured.
- **retEnd** – parameter that will get set to the actual index (in number of events) at which the end of the slice occured

Returns

A new EventStore object that is a shallow representation to the sliced, shared data. No data is copied over.

inline *AddressableEventStorage* **sliceTime**(const int64_t startTime, const int64_t endTime) const

Returns a new EventStore which is a shallow representation of a slice of this EventStore. The slice is from a specific startTime (in event timestamps, microseconds) to a specific endTime (event timestamps, microseconds). The actual size (in events) of the resulting packet depends on the event rate in the requested time interval. The resulting packet may be empty, if there is no event that happend in the requested interval.

As a slice is a shallow representation, no EventData gets copied by this operation. The resulting EventStore receives shared ownership over the relevant parts of the data. Should the original EventStore get out of scope, memory that is not relevant to the sliced EventStore will get freed.

The sliced output will be in the time range [startTime, endTime), endTime is exclusive.

Parameters

- **startTime** – The start time of the required slice (inclusive)
- **endTime** – The end time of the required time (exclusive)

Returns

A new EventStore object that is a shallow representation to the sliced, shared data. No data is copied over.

inline *AddressableEventStorage* **sliceBack**(const size_t length) const

Returns a new EventStore which is a shallow representation of a slice of this EventStore. Returns a slice which contains events from the back of the storage, it will contain no more events than given length variable.

As a slice is a shallow representation, no EventData gets copied by this operation. The resulting EventStore receives shared ownership over the relevant parts of the data. Should the original EventStore get out of scope, memory that is not relevant to the sliced EventStore will get freed.

Parameters

length – Maximum number of events contained in the resulting slice.

Returns

A new EventStore object that is a shallow representation to the sliced, shared data. No data is copied over.

inline *AddressableEventStorage* **sliceTime**(const int64_t startTime) const

Returns a new EventStore which is a shallow representation of a slice of this EventStore. The slice is from a specific startTime (in event timestamps, microseconds) to the end of the packet. The actual size (in events) of the resulting packet depends on the event rate in the requested time interval. The resulting packet may be empty, if there is no event that happened in the requested interval.

As a slice is a shallow representation, no EventData gets copied by this operation. The resulting EventStore receives shared ownership over the relevant parts of the data. Should the original EventStore get out of scope, memory that is not relevant to the sliced EventStore will get freed.

Parameters

startTime – The start time of the required slice, if positive. If negative, the number of microseconds from the end of the store

Returns

A new EventStore object that is a shallow representation to the sliced, shared data. No data is copied over.

inline *AddressableEventStorage* **sliceRate**(const double targetRate) const

Slices events from back of the EventStore, so that the EventStore would only contain a number of events of a given event rate. Useful for performance limited applications when it is required to limit the rate of events to maintain stable execution time.

Parameters

targetRate – Target event rate in events per second.

Returns

New event store which contains number of events within the target event rate.

inline *const_iterator* **begin**() const noexcept

Returns an iterator to the begin of the EventStore

Returns

an iterator to the begin of the EventStore

inline *const_iterator* **end**() const noexcept

Returns an iterator to the end of the EventStore

Returns

an iterator to the end of the EventStore

inline *const_reference* **front**() const

Returns a reference to the first element of the packet

Returns

a reference to the first element to the packet

inline *const_reference* **back**() const

Returns a reference to the last element of the packet

Returns

a reference to the last element to the packet

inline int64_t **getLowestTime**() const

Returns the timestamp of the first event in the packet. This is also the lowest timestamp in the packet, as the events are required to be monotonic.

Returns

The lowest timestamp present in the packet. 0 if the packet is empty.

inline int64_t **getHighestTime**() const

Returns the timestamp of the last event in the packet. This is also the highest timestamp in the packet, as the events are required to be monotonic.

Returns
The highest timestamp present in the packet. 0 if the packet is empty

inline size_t **getTotalLength**() const

Returns the total length (in number of events) of the packet

Returns
the total number of events present in the packet.

inline bool **isEmpty**() const

Returns true if the packet is empty (does not contain any events).

Returns
Returns true if the packet is empty (does not contain any events).

inline void **erase**(const size_t start, const size_t length)

Erase given range of events from the event store. This does not necessarily delete the underlying data since event store maps the data using smart pointers, the data will be cleared only in the case that none of the stores is mapping the data. This erase function does not affect data shared with other event stores.

Parameters

- **start** – Start index of events to erase
- **length** – Number of events to erase

inline size_t **eraseTime**(const int64_t startTime, const int64_t endTime)

Erase events in the range between given timestamps. This does not necessarily delete the underlying data since event store maps the data using smart pointers, the data will be cleared only in the case that none of the stores is mapping the data. This erase function does not affect data shared with other event stores.

Parameters

- **startTime** – Start timestamp for events to be erased, including this exact timestamp
- **endTime** – End timestamp for events to be erased, up to this time, events with this exact timestamp are not going to be erased.

Returns
Number of events deleted

inline const *EventType* &**operator[]**(const size_t index) const

Return an event at given index.

Parameters
index – Index of the event

Returns
Reference to the event at the index.

inline const *EventType* &**at**(const size_t index) const

Return an event at given index.

Parameters
index – Index of the event

Returns
Reference to the event at the index.

inline void **retainDuration**(const *dv*::*Duration* duration)

Retain a certain duration of event data in the event store. This will retain latest events and delete oldest data. The duration is just a hint of minimum amount of duration to keep, the exact duration will always be slightly greater (depending on event rate and memory allocation).

Parameters
duration – Minimum amount of time to keep in the event store. Events are erased in batches, so this guarantees only to maintain the batches of events within this duration.

inline *dv*::*Duration* **duration**() const

Get the duration of events contained.

Returns
Duration of stored events in microseconds.

inline bool **isWithinStoreTimeRange**(const int64_t timestamp) const

Checks whether given timestamp is within the time range of the event store.

Parameters
timestamp – Microsecond Unix timestamp to check.

Returns
True if the timestamp is within the time of event store, false otherwise.

inline size_t **getShardCapacity**() const

Get currently used default shard (data partial) capacity value.

Returns
Default capacity for new shards.

inline void **setShardCapacity**(const size_t shardCapacity)

Set a new capacity for shards (data partials). Setting this value does not affect already allocated shards and will be used only when a new shard needs to be allocated. If passed in capacity is set to 0, the setter will use a capacity value of 1, because that is the lowest allowed capacity value.

Parameters
shardCapacity – Capacity of events for newly allocated shards.

inline size_t **getShardCount**() const

Get the amount of shards that are currently referenced by the event store.

Returns
Number of referenced shards (data partials).

inline double **rate**() const

Get the event rate (events per second) for the events stored in this storage.

Returns
Events per second within this storage.

inline *EventPacketType* **toPacket**() const

Convert event store into a continuous memory packet. This performs a deep copy of underlying data.

Returns
Event packet with a copy of all stored events in this event store.

Protected Types

using **PartialEventDataType** = *PartialEventData*<*EventType*, *EventPacketType*>

Protected Functions

inline explicit **AddressableEventStorage**(const *std*::vector<*PartialEventDataType*> &dataPartials)

INTERNAL USE ONLY Creates a new EventStore based on the supplied `PartialEventData` objects. Offsets and meta information is recomputed from the supplied list. The packet gets shared ownership of all underlying data of the `PartialEventData` slices in `dataPartials`.

Parameters
dataPartials – vector of `PartialEventData` to construct this package from.

inline *PartialEventData*<*EventType*, *EventPacketType*> &**_getLastNonFullPartial**()

Retrieve the last partial that can store events. If available partial is full or no partials available at all, this function will instantiate, add the partial to the store, and return a reference to that partial.

Returns
Last data partial that can store an additional event.

Protected Attributes

std::vector<*PartialEventDataType*> **dataPartials_**

internal list of the shards.

std::vector<size_t> **partialOffsets_**

The exact number-of-events global offsets of the shards

size_t **totalLength_** = {0}

The total length of the event package

size_t **shardCapacity_** = {10000}

Default capacity for the data partials

Friends

friend class dv::io::MonoCameraWriter

friend class dv::io::NetworkWriter

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *AddressableEventStorage* &storage)

template<*concepts*::*AddressableEvent* **EventType**, class **EventPacketType**>

class **AddressableEventStorageIterator**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> Iterator for the EventStore class.

Public Types

using **iterator_category** = *std*::bidirectional_iterator_tag

using **value_type** = const *EventType*

using **pointer** = const *EventType**

using **reference** = const *EventType*&

using **size_type** = size_t

using **difference_type** = ptrdiff_t

Public Functions

inline **AddressableEventStorageIterator**()

Default constructor. Creates a new iterator at the beginning of the packet

inline explicit **AddressableEventStorageIterator**(const *std*::vector<*PartialEventData*<*EventType*, *EventPacketType*>> *dataPartialsPtr, const bool front)

Creates a new Iterator either at the beginning or at the end of the package

Parameters

- **dataPartialsPtr** – to the partials (shards) of the packet
- **front** – iterator will be at the beginning (true) of the packet, or at the end (false) of the packet.

inline **AddressableEventStorageIterator**(const *std*::vector<*PartialEventData*<*EventType*, *EventPacketType*>> *dataPartialsPtr, const size_t partialIndex, const size_t offset)

INTERNAL USE ONLY Creates a new iterator at the specific internal position supplied

Parameters

- **dataPartialsPtr** – Pointer to the partials (shards) of the packet
- **partialIndex** – Index pointing to the active shard
- **offset** – Offset in the active shard

inline *reference* **operator***() const noexcept

Returns
A reference to the Event at the current iterator position

inline *pointer* **operator->**() const noexcept

Returns
A pointer to the Event at current iterator position

inline *AddressableEventStorageIterator* &**operator++**() noexcept

Increments the iterator by one

Returns
A reference to the the same iterator, incremented by one

inline const *AddressableEventStorageIterator* **operator++**(int) noexcept

Post-increments the iterator by one

Returns
A new iterator at the current position. Increments original iterator by one.

inline *AddressableEventStorageIterator* &**operator+=**(const *size_type* add) noexcept

Increments iterator by a fixed number and returns reference to itself

Parameters
add – amount one whishes to increment the iterator

Returns
reference to itseld incremented by `add`

inline *AddressableEventStorageIterator* &**operator--**() noexcept

Decrements the iterator by one

Returns
A reference to the the same iterator, decremented by one

inline const *AddressableEventStorageIterator* **operator--**(int) noexcept

Post-decrement the iterator by one

Returns
A new iterator at the current position. Decrements original iterator by one.

inline *AddressableEventStorageIterator* &**operator-=**(const *size_type* sub) noexcept

Decrements iterator by a fixed number and returns reference to itself

Parameters
sub – amount one whishes to decrement the iterator

Returns
reference to itseld decremented by `sub`

inline bool **operator==**(const *AddressableEventStorageIterator* &rhs) const noexcept

Parameters
rhs – iterator to compare to

Returns
true if both iterators point to the same element

inline bool **operator!=**(const *AddressableEventStorageIterator* &rhs) const noexcept

Parameters
rhs – iterator to compare to

Returns
true if both iterators point to different elements

Private Functions

inline void **increment**()

Increments the iterator to the next event. If the iterator goes beyond available data, it remains at this position.

inline void **decrement**()

Decrements the iterator to the previous event. If the iterator goes below zero, it remains at zero.

Private Members

const *std*::vector<*PartialEventData*<*EventType*, *EventPacketType*>> ***dataPartialsPtr_**

size_t **partialIndex_**

The current partial (shard) we point to

size_t **offset_**

The current offset inside the shard we point to

template<class **EventStoreType**>

class **AddressableStereoEventStreamSlicer**

Public Functions

inline void **accept**(const *std*::optional<*EventStoreType*> &left, const *std*::optional<*EventStoreType*> &right)

Adds EventStores from the left and right camera. Performs job evaluation immediately.

Parameters

- **leftEvents** – the EventStore from left camera.
- **rightEvents** – the EventStore from right camera.

inline int **doEveryNumberOfEvents**(const size_t n, *std*::function<void(const *EventStoreType*&, const *EventStoreType*&)> callback)

Perform an action on the stereo stream data every given amount of events. Event count is evaluated on the left camera stream and according time interval of data is sliced from the right camera event stream. Sliced data is passed into the callback function as soon as it arrived, first argument is left camera events and second is right camera events. Since right camera events are sliced by the time interval of left camera, the amount of events on right camera can be different.

See also:

AddressableEventStreamSlicer::doEveryNumberOfEvents

Parameters

- **n** – the interval (in number of events) in which the callback should be called.
- **callback** – the callback function that gets called on the data every interval.

Returns

Job identifier

inline int **doEveryTimeInterval**(const *dv*::*Duration* interval, *std*::function<void(const *EventStoreType*&, const *EventStoreType*&)> callback)

Perform an action on the stereo stream data every given time interval. Event period is evaluated on the left camera stream and according time interval of data is sliced from the right camera event stream. Sliced data is passed into the callback function as soon as it arrived, first argument is left camera events and second is right camera events.

See also:

AddressableEventStreamSlicer::doEveryTimeInterval

Parameters

- **interval** – Time interval to call the callback function. The callback is called based on timestamps of left camera.
- **callback** – Function to be executed

Returns
Job identifier.

inline bool **hasJob**(const int job)

Returns true if the slicer contains the slicejob with the provided id

Parameters
job – the id of the slicejob in question

Returns
true, if the slicer contains the given slicejob

inline void **removeJob**(const int job)

Removes the given job from the list of current jobs.

Parameters
job – The job id to be removed

Protected Functions

inline void **clearRightEventsBuffer**(const int64_t timestampFrom)

Perform book-keeping of the right camera buffer by retaining data from a given timestamp. Events are "forgot" only if minimum amount and time duration values are maintained according to slicing configuration.

Parameters
timestampFrom – Perform book-keeping by retaining data from this timestamp onward.

Protected Attributes

std::optional<size_t> **minimumEvents** = *std*::nullopt

std::optional<*dv*::*Duration*> **minimumTime** = *std*::nullopt

StreamSlicer<*EventStoreType*> **slicer**

EventStoreType **leftEvents**

EventStoreType **rightEvents**

int64_t **rightEventSeek** = -1

struct **AedatFileError**

Public Types

using **Info** = *std*::filesystem::path

struct **AedatFileParseError**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **AedatVersionError**

Public Types

using **Info** = int32_t

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

template<*dv*::*concepts*::*TimeSurface*<*dv*::*EventStore*> **TimeSurface** = *dv*::*TimeSurface*, size_t **radius1** = 5, size_t **radius2** = 6>
class **ArcCornerDetector**

Public Types

using **UniquePtr** = *std*::unique_ptr<*ArcCornerDetector*>

using **SharedPtr** = *std*::shared_ptr<*ArcCornerDetector*>

Public Functions

ArcCornerDetector() = delete

template<typename ...**TIME_SURFACE_ADDITIONAL_ARGS**>
inline **ArcCornerDetector**(const cv::Size resolution, const typename *TimeSurface*::Scalar range, const bool resetTsAtEachIteration, *TIME_SURFACE_ADDITIONAL_ARGS*&&... timeSurfaceAdditionalArgs)

Constructor

Template Parameters
TIME_SURFACE_ADDITIONAL_ARGS – Types of the additional arguments passed to the time surface constructor

Parameters

- **resolution** – camera dimensions
- **range** – the range within which the timestamps of a corner should be for it to be detected as a corner
- **resetTsAtEachIteration** – set to true if the time surface should be reset at each iteration
- **timeSurfaceAdditionalArgs** – arguments passed to the time surface constructor in addition to the resolution

inline *dv*::*cvector*<*dv*::*TimedKeyPoint*> **detect**(const *dv*::*EventStore* &events, const cv::Rect &roi, const cv::Mat &mask)

Runs the detection algorithm.

A corner is defined by two arcs of different radii containing timestamps which satisfy the following conditions:

- All timestamps that are on the corner are within a range of mCornerRange.
- No timestamp that is outside of this corner is greater than or equal to the minimum timestamp within the corner
- Length of the arc is within the ranges [ArcLimits::MIN_ARC_SIZE_FACTOR * circumference, ArcLimits::MAX_ARC_SIZE_FACTOR * circumference].

See also:

ArcLimits.

Parameters

- **events** – events
- **roi** – region of interest
- **mask** – mask containing zeros for all pixels which should be ignored and nonzero for all others

Returns
a vector containing the detected keypoints. The response is defined as the difference between the minimum timestamp within the arc and the maximum timestamp outside of the arc.

inline auto **getTimeSurface**(const bool polarity) const

Returns the TimeSurface for a given polarity

Parameters
polarity – the polarity

Returns
the requested time surface

Private Functions

inline auto **insideCorner**(const int64_t ts1, const int64_t ts2)

template<typename **ITERATOR**>
inline auto **expandArc**(const *ITERATOR* &maxTimestampLoc, const int64_t maxTimestampValue, const *dv*::Event &event, const CircularTimeSurfaceView &circle)

template<typename **ITERATOR**>
inline auto **checkSurroundingTimestamps**(const *ITERATOR* &arcBegin, const *ITERATOR* arcEnd, const int64_t minTimestampInArc, const *dv*::Event &event, const CircularTimeSurfaceView &circle)

Private Members

std::array<*TimeSurface*, 2> **mTimeSurfaces**

int64_t **mCornerRange**

bool **mResetTsAfterDetection**

std::array<CircularTimeSurfaceView, 2> **mCircles**

std::array<ArcLimits, 2> **mArcLimits**

class **ArcLimits**

Public Functions

inline explicit **ArcLimits**(const size_t circumference)

inline auto **satisfied**(const size_t arcSize) const

Private Members

const size_t **mCircumference**

const size_t **mMinSize**

const size_t **mMaxSize**

Private Static Attributes

static constexpr float **MIN_ARC_SIZE_FACTOR** = 0.125f

static constexpr float **MAX_ARC_SIZE_FACTOR** = 0.4f

template<class **EventStoreClass** = *dv*::*EventStore*>

class **BackgroundActivityNoiseFilter** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

Public Functions

inline explicit **BackgroundActivityNoiseFilter**(const cv::Size &resolution, const *dv*::*Duration* backgroundActivityDuration = *dv*::*Duration*(2000))

Initiate a background activity noise filter, which test the neighbourhoods of incoming events for other supporting events that happened within the background activity period.

Parameters

- **resolution** – Sensor resolution.
- **backgroundActivityDuration** – Background activity duration.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &evt) noexcept override

Test the background activity, if the event neighbourhood has at least one event that was triggered within the background activity duration, the event will not be considered noise and should be retained, and discarded otherwise.

Parameters

evt – Event to be checked.

Returns

True to retain event, false to discard.

inline *BackgroundActivityNoiseFilter* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters

events – Input events.

Returns

inline *dv*::*Duration* **getBackgroundActivityDuration**() const

Get currently configured background activity duration value.

Returns

Background activity duration value.

inline void **setBackgroundActivityDuration**(const *dv*::*Duration* backgroundActivityDuration)

Set new background activity duration value.

Parameters

backgroundActivityDuration – Background activity duration value.

Protected Functions

inline bool **doBackgroundActivityLookup_unsafe**(int16_t x, int16_t y, int64_t timestamp)

inline bool **doBackgroundActivityLookup**(int16_t x, int16_t y, int64_t timestamp)

Protected Attributes

cv::Size **mResolutionLimits**

dv::*TimeSurface* **mTimeSurface**

int64_t **mBackgroundActivityDuration** = 2000

struct **BadAlloc** : public *dv*::*exceptions*::*info*::*EmptyException*

template<class **T**>

class **basic_cstring**

Public Types

using **value_type** = *T*

using **const_value_type** = const *T*

using **pointer** = *T**

using **const_pointer** = const *T**

using **reference** = *T*&

using **const_reference** = const *T*&

using **size_type** = size_t

using **difference_type** = ptrdiff_t

using **iterator** = *cPtrIterator*<*value_type*>

using **const_iterator** = *cPtrIterator*<*const_value_type*>

using **reverse_iterator** = *std*::reverse_iterator<*iterator*>

using **const_reverse_iterator** = *std*::reverse_iterator<*const_iterator*>

Public Functions

constexpr **basic_cstring**() noexcept = default

inline **~basic_cstring**() noexcept

inline **basic_cstring**(const *basic_cstring* &str, const *size_type* pos = 0, const *size_type* count = *npos*)

constexpr **basic_cstring**(*std*::nullptr_t) = delete

inline **basic_cstring**(*const_pointer* str)

template<typename **U**>
inline **basic_cstring**(const *U* &str, const *size_type* pos = 0, const *size_type* count = *npos*)

inline **basic_cstring**(*const_pointer* str, const *size_type* strLength, const *size_type* pos = 0, const *size_type* count = *npos*)

inline explicit **basic_cstring**(const *size_type* count)

inline **basic_cstring**(const *size_type* count, const *value_type* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline **basic_cstring**(*InputIt* first, *InputIt* last)

inline **basic_cstring**(*std*::initializer_list<*value_type*> init_list)

template<typename **U** = *T*, *std*::enable_if_t<*std*::is_same_v<*U*, char>, bool> = true>
inline **basic_cstring**(const *std*::filesystem::path &path)

template<typename **U** = *T*, *std*::enable_if_t<*std*::is_same_v<*U*, wchar_t>, bool> = true>
inline **basic_cstring**(const *std*::filesystem::path &path)

template<typename **U** = *T*, *std*::enable_if_t<*std*::is_same_v<*U*, char8_t>, bool> = true>

inline **basic_cstring**(const *std*::filesystem::path &path)

template<typename **U** = *T*, *std*::enable_if_t<*std*::is_same_v<*U*, char16_t>, bool> = true>
inline **basic_cstring**(const *std*::filesystem::path &path)

template<typename **U** = *T*, *std*::enable_if_t<*std*::is_same_v<*U*, char32_t>, bool> = true>
inline **basic_cstring**(const *std*::filesystem::path &path)

inline **basic_cstring**(*basic_cstring* &&rhs) noexcept

inline *basic_cstring* &**operator=**(*basic_cstring* &&rhs) noexcept

inline *basic_cstring* &**operator=**(const *basic_cstring* &rhs)

inline *basic_cstring* &**operator=**(*const_pointer* str)

template<typename **U**>
inline *basic_cstring* &**operator=**(const *U* &rhs)

inline *basic_cstring* &**operator=**(const *value_type* value)

inline *basic_cstring* &**operator=**(*std*::initializer_list<*value_type*> rhs_list)

inline bool **operator==**(const *basic_cstring* &rhs) const noexcept

inline auto **operator<=>**(const *basic_cstring* &rhs) const noexcept

inline bool **operator==**(*const_pointer* rhs) const noexcept

inline auto **operator<=>**(*const_pointer* rhs) const noexcept

template<typename **U**>
inline bool **operator==**(const *U* &rhs) const noexcept

template<typename **U**>
inline auto **operator<=>**(const *U* &rhs) const noexcept

inline *basic_cstring* &**assign**(*basic_cstring* &&str)

inline *basic_cstring* &**assign**(const *basic_cstring* &str, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *basic_cstring* &**assign**(*const_pointer* str)

template<typename **U**>
inline *basic_cstring* &**assign**(const *U* &str, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *basic_cstring* &**assign**(*const_pointer* str, const *size_type* strLength, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *basic_cstring* &**assign**(const *value_type* value)

inline *basic_cstring* &**assign**(const *size_type* count, const *value_type* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline *basic_cstring* &**assign**(*InputIt* first, *InputIt* last)

inline *basic_cstring* &**assign**(*std*::initializer_list<*value_type*> init_list)

inline *pointer* **data**() noexcept

inline *const_pointer* **data**() const noexcept

inline *const_pointer* **c_str**() const noexcept

inline *size_type* **size**() const noexcept

inline *size_type* **length**() const noexcept

inline *size_type* **capacity**() const noexcept

inline *size_type* **max_size**() const noexcept

inline bool **empty**() const noexcept

inline void **resize**(const *size_type* newSize)

inline void **resize**(const *size_type* newSize, const *value_type* value)

inline void **reserve**(const *size_type* minCapacity)

inline void **shrink_to_fit**()

template<typename **INT**>
inline *reference* **operator[]**(const *INT* index)

template<typename **INT**>
inline *const_reference* **operator[]**(const *INT* index) const

template<typename **INT**>
inline *reference* **at**(const *INT* index)

template<typename **INT**>
inline *const_reference* **at**(const *INT* index) const

inline **operator** ***std*****::basic_string_view<*****value_type*****>**() const

inline explicit **operator** ***std*****::basic_string<*****value_type*****>**() const

inline *reference* **front**()

inline *const_reference* **front**() const

inline *reference* **back**()

inline *const_reference* **back**() const

inline void **push_back**(const *value_type* value)

inline void **pop_back**()

inline void **clear**() noexcept

inline void **swap**(*basic_cstring* &rhs) noexcept

inline *iterator* **begin**() noexcept

inline *iterator* **end**() noexcept

inline *const_iterator* **begin**() const noexcept

inline *const_iterator* **end**() const noexcept

inline *const_iterator* **cbegin**() const noexcept

inline *const_iterator* **cend**() const noexcept

inline *reverse_iterator* **rbegin**() noexcept

inline *reverse_iterator* **rend**() noexcept

inline *const_reverse_iterator* **rbegin**() const noexcept

inline *const_reverse_iterator* **rend**() const noexcept

inline *const_reverse_iterator* **crbegin**() const noexcept

inline *const_reverse_iterator* **crend**() const noexcept

inline *iterator* **insert**(*const_iterator* pos, const *value_type* value)

inline *iterator* **insert**(*const_iterator* pos, const *size_type* count, const *value_type* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline *iterator* **insert**(*const_iterator* pos, *InputIt* first, *InputIt* last)

inline *iterator* **insert**(*const_iterator* pos, *std*::initializer_list<*value_type*> init_list)

inline *iterator* **erase**(*const_iterator* pos)

inline *iterator* **erase**(*const_iterator* first, *const_iterator* last)

inline constexpr *size_type* **find**(const *basic_cstring* &str, *size_type* pos = 0) const noexcept

inline constexpr *size_type* **find**(*value_type* c, *size_type* pos = 0) const noexcept

inline constexpr *size_type* **find**(*const_pointer* s, *size_type* pos, *size_type* count) const

inline constexpr *size_type* **find**(*const_pointer* s, *size_type* pos = 0) const

inline constexpr *size_type* **rfind**(const *basic_cstring* &str, *size_type* pos = *npos*) const noexcept

inline constexpr *size_type* **rfind**(*value_type* c, *size_type* pos = *npos*) const noexcept

inline constexpr *size_type* **rfind**(*const_pointer* s, *size_type* pos, *size_type* count) const

inline constexpr *size_type* **rfind**(*const_pointer* s, *size_type* pos = *npos*) const

inline *basic_cstring* &**append**(const *basic_cstring* &str, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *basic_cstring* &**append**(*const_pointer* str)

template<typename **U**>
inline *basic_cstring* &**append**(const *U* &str, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *basic_cstring* &**append**(*const_pointer* str, const *size_type* strLength, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *basic_cstring* &**append**(const *value_type* value)

inline *basic_cstring* &**append**(const *size_type* count, const *value_type* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline *basic_cstring* &**append**(*InputIt* first, *InputIt* last)

inline *basic_cstring* &**append**(*std*::initializer_list<*value_type*> init_list)

inline *basic_cstring* &**operator+=**(const *basic_cstring* &rhs)

inline *basic_cstring* &**operator+=**(*const_pointer* str)

template<typename **U**>
inline *basic_cstring* &**operator+=**(const *U* &str)

inline *basic_cstring* &**operator+=**(const *value_type* value)

inline *basic_cstring* &**operator+=**(*std*::initializer_list<*value_type*> rhs_list)

inline *basic_cstring* **operator+**(const *basic_cstring* &rhs) const

inline *basic_cstring* **operator+**(*const_pointer* rhs) const

template<typename **U**>
inline *basic_cstring* **operator+**(const *U* &rhs) const

inline *basic_cstring* **operator+**(const *value_type* value) const

inline *basic_cstring* **operator+**(*std*::initializer_list<*value_type*> rhs_list) const

Public Static Attributes

static constexpr *size_type* **npos** = {static_cast<*size_type*>(-1)}

Private Functions

inline void **nullTerminate**()

inline void **ensureCapacity**(const *size_type* newSize)

inline void **reallocateMemory**(const *size_type* newSize)

inline *size_type* **getIndex**(const *size_type* index) const

inline *size_type* **getIndex**(const *difference_type* index) const

Private Members

size_type **mCurrSize** = {0}

size_type **mMaximumSize** = {0}

pointer **mDataPtr** = {&*NULL_CHAR*}

Private Static Attributes

static *T* **NULL_CHAR** = {0}

Friends

inline friend *basic_cstring* **operator+** (*const_pointer* lhs, const *basic_cstring* &rhs)

template<typename **U**>
inline friend *basic_cstring* **operator+** (const *U* &lhs, const *basic_cstring* &rhs)

inline friend *basic_cstring* **operator+** (const *value_type* value, const *basic_cstring* &rhs)

inline friend *basic_cstring* **operator+** (*std*::initializer_list<*value_type*> lhs_list, const *basic_cstring* &rhs)

inline friend *std*::ostream &**operator<<** (*std*::ostream &os, const *basic_cstring* &rhs)

struct **BoundingBox** : public *flatbuffers*::NativeTable

Public Types

typedef *BoundingBoxFlatbuffer* **TableType**

Public Functions

inline **BoundingBox** ()

inline **BoundingBox** (int64_t _timestamp, float _topLeftX, float _topLeftY, float _bottomRightX, float _bottomRightY, float _confidence, const *dv*::*cstring* &_label)

Public Members

int64_t **timestamp**

float **topLeftX**

float **topLeftY**

float **bottomRightX**

float **bottomRightY**

float **confidence**

dv::*cstring* **label**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

struct **BoundingBoxBuilder**

Public Functions

inline void **add_timestamp**(int64_t timestamp)

inline void **add_topLeftX**(float topLeftX)

inline void **add_topLeftY**(float topLeftY)

inline void **add_bottomRightX**(float bottomRightX)

inline void **add_bottomRightY**(float bottomRightY)

inline void **add_confidence**(float confidence)

inline void **add_label**(*flatbuffers*::Offset<*flatbuffers*::String> label)

inline explicit **BoundingBoxBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

BoundingBoxBuilder &**operator=**(const *BoundingBoxBuilder*&)

inline *flatbuffers*::Offset<*BoundingBoxFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **BoundingBoxFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *BoundingBox* **NativeTableType**

Public Functions

inline int64_t **timestamp**() const
Timestamp (µs).

inline float **topLeftX**() const
top left corner of bounding box x-coordinate.

inline float **topLeftY**() const
top left corner of bounding box y-coordinate.

inline float **bottomRightX**() const
bottom right corner of bounding box x-coordinate.

inline float **bottomRightY**() const
bottom right corner of bounding box y-coordinate.

inline float **confidence**() const
confidence of the given bounding box.

inline const *flatbuffers*::String ***label**() const
Label for the given bounding box.

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *BoundingBox* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*BoundingBox* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*BoundingBox* *_o, const *BoundingBoxFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*BoundingBoxFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *BoundingBox* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

struct **BoundingBoxPacket** : public *flatbuffers*::NativeTable

Public Types

typedef *BoundingBoxPacketFlatbuffer* **TableType**

Public Functions

inline **BoundingBoxPacket**()

inline **BoundingBoxPacket**(const *dv*::*cvector*<*BoundingBox*> &_elements)

Public Members

dv::*cvector*<*BoundingBox*> **elements**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *BoundingBoxPacket* &packet)

struct **BoundingBoxPacketBuilder**

Public Functions

inline void **add_elements**(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*BoundingBoxFlatbuffer*>>> elements)

inline explicit **BoundingBoxPacketBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

BoundingBoxPacketBuilder &**operator=**(const *BoundingBoxPacketBuilder*&)

inline *flatbuffers*::Offset<*BoundingBoxPacketFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **BoundingBoxPacketFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *BoundingBoxPacket* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*BoundingBoxFlatbuffer*>> ***elements**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *BoundingBoxPacket* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*BoundingBoxPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*BoundingBoxPacket* *_o, const *BoundingBoxPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*BoundingBoxPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *BoundingBoxPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "BBOX"

class **CalibrationSet**

#include </builds/inivation/dv/dv-processing/include/dv-processing/camera/calibration_set.hpp> *CalibrationSet* class is used to store, serialize and deserialize various camera related calibrations - intrinsic, extrinsic, *IMU* calibrations. Supports multi-camera and multi sensor setups.

Each calibration for each sensor received a designation string which consist of a letter determining the type of sensor and a numeric index automatically generated for each sensor. Designation string look like this: “C0” - camera with

index 0 “S0” - *IMU* sensor with index 0 “C0C1” - stereo calibration where C0 is the left camera and C1 is the right camera in the camera rig setup.

Designation indexes are automatically incremented by the order they are added to the calibration set.

Public Functions

CalibrationSet() = default

inline pt::ptree **toPropertyTree**() const

Serialize calibration data into a property tree that can be saved into a file using `boost::property_tree::write_json` or other `property_tree` serialization method.

Returns
Property tree containing calibration data.

inline *std*::vector<*std*::string> **getCameraList**() const

Get a list of cameras available by their designation.

Returns
Vector of available camera designations.

inline *std*::vector<*std*::string> **getImuList**() const

Get a list camera designations which have imu calibrations available in this calibration set.

Returns
Vector of available imu designations.

inline *std*::vector<*std*::string> **getStereoList**() const

Get a list of designations of stereo calibrations available here.

Returns
Vector of available stereo calibrations designations.

inline *std*::optional<*calibrations*::*CameraCalibration*> **getCameraCalibration**(const *std*::string &designation) const

Retrieve a camera calibration by designation (e.g. “C0”).

Designation string consists of a letter determining the type of sensor and a numeric index automatically generated for each sensor. Designation string look like this: “C0” - camera with index 0 “S0” - *IMU* sensor with index 0 “C0C1” - stereo calibration where C0 is the left camera and C1 is the right camera in the camera rig setup.

Parameters
designation – Camera designation string.

Returns
Camera instrinsics calibration, `std::nullopt` if given designation is not found.

inline *std*::optional<*calibrations*::*IMUCalibration*> **getImuCalibration**(const *std*::string &designation) const

Get *IMU* calibration by *IMU* sensor designation (e.g. “S0”).

Designation string consists of a letter determining the type of sensor and a numeric index automatically generated for each sensor. Designation string look like this: “C0” - camera with index 0 “S0” - *IMU* sensor with index 0 “C0C1” - stereo calibration where C0 is the left camera and C1 is the right camera in the camera rig setup.

Parameters
designation – *IMU* designation string.

Returns
IMU extrinsic calibration, `std::nullopt` if given designation is not found.

inline *std*::optional<*calibrations*::*StereoCalibration*> **getStereoCalibration**(const *std*::string &designation) const

Get stereo calibration by stereo rig designation (e.g. "C0C1").Retrieve the full list of *IMU* extrinsic calibrations.

Designation string consists of a letter determining the type of sensor and a numeric index automatically generated for each sensor. Designation string look like this: "C0" - camera with index 0 "S0" - *IMU* sensor with index 0 "C0C1" - stereo calibration where C0 is the left camera and C1 is the right camera in the camera rig setup.

Parameters
designation – Stereo rig designation string.

Returns
Stereo extrinsic calibration, `std::nullopt` if given designation is not found.

inline *std*::optional<*calibrations*::*CameraCalibration*> **getCameraCalibrationByName**(const *std*::string &camera) const

Retrieve a camera calibration by camera name, which consist of model and serial number concatenation with an underscore separator (e.g. "DVXplorer_DXA00000").

Camera name is usually available in recording files and when connected directly to a camera.

Parameters
camera – Name of the camera.

Returns
Camera intrinsic calibration, `std::nullopt` if given camera name is not found.

inline *std*::optional<*calibrations*::*IMUCalibration*> **getImuCalibrationByName**(const *std*::string &camera) const

Retrieve an *IMU* calibration by camera name, which consist of model and serial number concatenation with an underscore separator (e.g. "DVXplorer_DXA00000").

Camera name is usually available in recording files and when connected directly to a camera.

Parameters
camera – Name of the camera.

Returns
IMU extrinsics calibration, `std::nullopt` if given camera name is not found.

inline *std*::optional<*calibrations*::*StereoCalibration*> **getStereoCalibrationByLeftCameraName**(const *std*::string &camera) const

Retrieve a stereo calibration by matching camera name to left camera name in the stereo calibrations. Camera name consist of model and serial number concatenation with an underscore separator (e.g. "DVXplorer_DXA00000").

Camera name is usually available in recording files and when connected directly to a camera.

Parameters
camera – Name of the camera.

Returns
Stereo extrinsic calibration, `std::nullopt` if given camera name is not found.

inline *std*::optional<*calibrations*::*StereoCalibration*> **getStereoCalibrationByRightCameraName**(const *std*::string &camera) const

Retrieve a stereo calibration by matching camera name to right camera name in the stereo calibrations. Camera name consist of model and serial number concatenation with an underscore separator (e.g. "DVXplorer_DXA00000").

Camera name is usually available in recording files and when connected directly to a camera.

Parameters
camera – Name of the camera.

Returns
Stereo extrinsic calibration, `std::nullopt` if given camera name is not found.

inline void **updateImuCalibration**(const *calibrations*::*IMUCalibration* &calibration)

Update *IMU* calibration for the camera name.

Parameters
calibration – *IMU* calibration instance.

inline void **updateCameraCalibration**(const *calibrations*::*CameraCalibration* &calibration)

Update Camera calibration for the given camera name.

Parameters
calibration – Camera calibration instance.

inline void **updateStereoCameraCalibration**(const *calibrations*::*StereoCalibration* &calibration)

Update Stereo Camera calibration for the given camera name.

Parameters
calibration – Stereo calibration instance.

inline void **addCameraCalibration**(const *calibrations*::*CameraCalibration* &calibration)

Add an intrinsic calibration to the camera calibration set. Camera designation is going to be generated automatically.

Parameters
calibration – Camera intrinsics calibration.

inline void **addImuCalibration**(const *calibrations*::*IMUCalibration* &calibration)

Add an *IMU* extrinsics calibration to the calibration set.

Parameters
calibration – *IMU* extrinsic calibration.

inline void **addStereoCalibration**(const *calibrations*::*StereoCalibration* &calibration)

Add a stereo calibration to the calibration set. Intrinsic calibrations of the sensors should already be added using `addCameraCalibration` prior to adding the stereo extrinsic calibration.

Parameters
calibration – Stereo calibration.

Throws
`Throws` – an invalid argument exception if the intrinsic calibration of given camera sensors are not available in the set or stereo calibration for the given cameras already exist/

inline const *std*::map<*std*::string, *calibrations*::*CameraCalibration*> &**getCameraCalibrations**() const

Retrieve the full list of camera intrinsic calibrations.

Returns
`std::map` containing camera calibrations where keys are camera designation strings.

inline const *std*::map<*std*::string, *calibrations*::*IMUCalibration*> &**getImuCalibrations**() const

Retrieve the full list of *IMU* extrinsic calibrations.

Returns
`std::map` containing *IMU* calibrations where keys are *IMU* sensor designation strings.

inline const *std*::map<*std*::string, *calibrations*::*StereoCalibration*> &**getStereoCalibrations**() const

Retrieve the full list of stereo extrinsic calibrations.

Returns
`std::map` containing stereo calibrations where keys are stereo rig camera designation strings.

inline void **writeToFile**(const fs::path &outputFile) const

Write the contents of this calibration set into a file at given path.

This function requires that supplied path contains ".json" extension.

Parameters
outputFile – Output file path with ".json" extension to write the contents of the calibration set.

Public Static Functions

static inline *CalibrationSet* **LoadFromFile**(const fs::path &path)

Create a calibration file representation from a persistent file. Supports legacy ".xml" calibration files produced by DV as well as JSON files containing calibration of a new format.

The file format is distinguished using the file path extension.

Parameters
path – Path to calibration file.

Returns
CalibrationFile instanced containing parsed calibration values.

Public Static Attributes

static constexpr *std*::array<float, 16> **identity**{1.f, 0.f, 0.f, 0.f, 0.f, 1.f, 0.f, 0.f, 0.f, 0.f, 1.f, 0.f, 0.f, 0.f, 0.f, 1.f}

Private Types

using **CameraCalibrationMap** = *std*::map<*std*::string, *calibrations*::*CameraCalibration*>

using **IMUCalibrationMap** = *std*::map<*std*::string, *calibrations*::*IMUCalibration*>

using **StereoCalibrationMap** = *std*::map<*std*::string, *calibrations*::*StereoCalibration*>

Private Functions

inline explicit **CalibrationSet**(const pt::ptree &tree)

Private Members

size_t **cameraIndex** = 0

size_t **imuIndex** = 0

CameraCalibrationMap **cameras**

IMUCalibrationMap **imus**

StereoCalibrationMap **stereo**

Private Static Functions

static inline *CalibrationSet* **cameraRigCalibrationFromJsonFile**(const fs::path &path)

static inline *calibrations*::*CameraCalibration* **oneCameraCalibrationFromXML**(const cv::FileNode &node, const *std*::string_view cameraName, const bool cameraIsMaster)

static inline *CalibrationSet* **cameraRigCalibrationFromXmlFile**(const fs::path &path)

class **CameraCalibration**

Public Functions

CameraCalibration() = default

inline explicit **CameraCalibration**(const pt::ptree &tree)

Parse a property tree and initialize camera calibration out of it.

Parameters

tree – Serialized property tree containing camera intrinsics calibration.

inline **CameraCalibration**(const *std*::string_view name_, const *std*::string_view position_, const bool master_, const cv::Size &resolution_, const cv::Point2f &principalPoint_, const cv::Point2f &focalLength_, const *std*::vector<float> &distortion_, const *DistortionModel* &distortionModel_, *std*::span<const float> transformationToC0View, const *std*::optional<*Metadata*> &metadata_)

Construct the camera calibration

Parameters

- **name_** – Camera name (e.g. "DVXplorer_DXA02137")
- **position_** – Description of the location of the camera in the camera rig (e.g. "left")
- **master_** – Whether camera was a master camera during calibration
- **resolution_** – Camera resolution
- **principalPoint_** – Principal point
- **focalLength_** – Focal length
- **distortion_** – Distortion coefficients
- **distortionModel_** – Distortion model used (can be empty string or "radialTangential")
- **transformationToC0_** – Transformation from camera zero to this camera
- **metadata_** – *Metadata*

inline pt::ptree **toPropertyTree**() const

Serialize the *CameraCalibration* structure into a property tree.

Returns

Serialized property tree.

inline Eigen::Matrix4f **getTransformMatrix**() const

Return the transformation matrix to C0 as a Eigen matrix.

Returns

Eigen matrix containing transformation to camera "C0".

inline cv::Matx33f **getCameraMatrix**() const

Get camera matrix in the format: | mFx 0 mCx | | 0 mFy mCy | | 0 0 1 | for direct OpenCV compatibility.

Returns

3x3 Camera matrix with pixel length values

inline bool **operator==**(const *CameraCalibration* &rhs) const

Equality operator for the class, compares each member of the class.

Parameters

rhs – Other instance of this class

Returns

inline *dv*::*camera*::*CameraGeometry* **getCameraGeometry**() const

Retrieve camera geometry instance from this calibration instance. Distortion model is going to be ignored if the `CameraGeometry` class doesn't support the distortion model.

CameraGeometry class only supports "radialTangential" distortion model.

Returns
Camera geometry class that implements geometrical transformations of pixel coordinates.

inline *std*::string **getDistortionModelString**() const

Get distortion model name as a string.

Returns
Distortion model name.

Public Members

std::string **name**

Camera name (e.g. "DVXplorer_DXA02137")

std::string **position**

Description of the location of the camera in the camera rig (e.g. "left")

bool **master** = false

Indicate whether it is the master camera in a multi-camera rig.

cv::Size **resolution**

Camera resolution width.

cv::Point2f **principalPoint**

Intersection of optical axis and image plane.

cv::Point2f **focalLength**

Focal length.

std::vector<float> **distortion**

Distortion coefficients.

DistortionModel **distortionModel** = *DistortionModel*::*RadTan*

Distortion model used.

std::vector<float> **transformationToC0**

Transformation from camera zero to this camera.

std::optional<*Metadata*> **metadata** = *std*::nullopt

Metadata.

Protected Static Functions

template<typename **T**>
static inline void **pushVectorToTree**(const *std*::string &key, const *std*::vector<*T*> &vals, pt::ptree &tree)

Push a vector of the given type to the property tree at the given key.

template<typename **T**>
static inline *std*::vector<*T*> **getVectorFromTree**(const *std*::string &key, const pt::ptree &tree)

Retrieve a vector of the given type from the property tree from the given key.

Returns
A sequence value in a std::vector container.

template<class **Container**, typename **Scalar**>
static inline *Container* **parsePair**(const pt::ptree &child, const *std*::string &name, *std*::optional<*Scalar*> defaults = *std*::nullopt)

template<class **Container**, typename **Scalar**>
static inline *Container* **parseTripple**(const pt::ptree &child, const *std*::string &name, *std*::optional<*Scalar*> defaults = *std*::nullopt)

template<class **MetadataClass**>
static inline *std*::optional<*MetadataClass*> **getOptionalMetadata**(const boost::property_tree::ptree &tree, const *std*::string &path)

static inline bool **homogeneityCheck**(const *std*::vector<float> &transformation)

static inline void **validateTransformation**(const *std*::vector<float> &transformation)

Friends

friend struct IMUCalibration

friend struct StereoCalibration

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *CameraCalibration* &calibration)

Serialize the object into a stream.

Parameters

- **os** –
- **calibration** –

Returns

class **CameraCapture** : public *dv*::*io*::*CameraInputBase*

Public Types

enum class **BiasSensitivity**

Values:

enumerator **VeryLow**

enumerator **Low**

enumerator **Default**

enumerator **High**

enumerator **VeryHigh**

enum class **DavisReadoutMode**

Values:

enumerator **EventsAndFrames**

enumerator **EventsOnly**

enumerator **FramesOnly**

enum class **DavisColorMode**

Values:

enumerator **Grayscale**

enumerator **Color**

enum class **DVXeFPS**

Values:

enumerator **EFPS_CONSTANT_100**

enumerator **EFPS_CONSTANT_200**

enumerator **EFPS_CONSTANT_500**

enumerator **EFPS_CONSTANT_1000**

enumerator **EFPS_CONSTANT_LOSSY_2000**

enumerator **EFPS_CONSTANT_LOSSY_5000**

enumerator **EFPS_CONSTANT_LOSSY_10000**

enumerator **EFPS_VARIABLE_2000**

enumerator **EFPS_VARIABLE_5000**

enumerator **EFPS_VARIABLE_10000**

enumerator **EFPS_VARIABLE_15000**

enum class **CameraType**

Values:

enumerator **Any**

enumerator **DAVIS**

enumerator **DVS**

Public Functions

inline **CameraCapture**()

Create a camera capture class which opens first discovered camera of any type.

inline explicit **CameraCapture**(const *std*::string &cameraName, const *CameraType* type = *CameraType*::*Any*)

Create a camera capture class which opens a camera according to given parameters.

Parameters

- **cameraName** – Camera name, an empty string will match any name.
- **type** – Type of camera, one of: any, DVS, or DAVIS.

inline virtual *std*::optional<*dv*::*EventStore*> **getNextEventBatch**() override

Parse and retrieve next event batch.

Returns

Event batch or `std::nullopt` if no events were received since last read.

inline virtual *std*::optional<*dv*::*Frame*> **getNextFrame**() override

Parse and retrieve next frame.

Returns

Frame or `std::nullopt` if no frames were received since last read.

inline virtual *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **getNextImuBatch**() override

Parse and retrieve next *IMU* data batch.

Returns

IMU data batch or `std::nullopt` if no *IMU* data was received since last read.

inline virtual *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **getNextTriggerBatch**() override

Parse and retrieve next trigger data batch.

Returns

Trigger data batch or `std::nullopt` if no triggers were received since last read.

inline virtual *std*::optional<cv::Size> **getEventResolution**() const override

Get event stream resolution.

Returns

Event stream resolution, `std::nullopt` if event stream is unavailable.

inline virtual *std*::optional<cv::Size> **getFrameResolution**() const override

Retrieve frame stream resolution.

Returns

Frame stream resolution or `std::nullopt` if the frame stream is not available.

inline virtual bool **isFrameStreamAvailable**() const override

Check whether frame stream is available.

Returns

True if frame stream is available, false otherwise.

inline virtual bool **isEventStreamAvailable**() const override

Check whether event stream is available.

Returns

True if event stream is available, false otherwise.

inline virtual bool **isImuStreamAvailable**() const override

Check whether device outputs *IMU* data.

Returns

True if device outputs *IMU* data, false otherwise.

inline virtual bool **isTriggerStreamAvailable**() const override

Check whether device outputs trigger data.

Returns

True if device outputs trigger data, false otherwise.

inline **~CameraCapture**()

Destructor: stops the readout thread.

inline virtual *std*::string **getCameraName**() const override

Get camera name, which is a combination of the camera model and the serial number.

Returns

String containing the camera model and serial number separated by an underscore character.

inline bool **enableDavisAutoExposure**()

Enable auto-exposure. To disable the auto-exposure, use the manual set exposure function.

Returns

True if configuration was successful, false otherwise.

inline bool **setDavisExposureDuration**(const *dv*::*Duration* &exposure)

Disable auto-exposure and set a new fixed exposure value.

Parameters
exposure – Exposure duration.

Returns
True if configuration was successful, false otherwise.

inline *std*::optional<*dv*::*Duration*> **getDavisExposureDuration**() const

Get the current exposure duration.

Returns
An optional containing the exposure duration, return `std::nullopt` in case exposure duration setting is not available for the device.

inline bool **setDavisFrameInterval**(const *dv*::*Duration* &interval)

Set a new frame interval value. This interval defines the framerate output of the camera. The frames will be produced at the given interval, the interval can be reduced in case exposure time is longer than the frame interval.

Parameters
interval – Output frame interval.

Returns
True if configuration was successful, false otherwise.

inline *std*::optional<*dv*::*Duration*> **getDavisFrameInterval**() const

Get the configured frame interval.

Returns
An optional containing the frame interval value, return `std::nullopt` in case frame interval setting is not available for the device.

inline uint32_t **deviceConfigGet**(const int8_t moduleAddress, const uint8_t parameterAddress) const

Get a configuration setting value from the connected device.

Parameters

- **moduleAddress** – Module address. An integer number that represents a group of settings.
- **parameterAddress** – Parameter address. An integer number that specifies a parameter within a parameter module group.

Throws
`runtime_error` – Exception is thrown if parameter is not available for the device.

Returns
Configured value of the parameter.

inline void **deviceConfigSet**(const int8_t moduleAddress, const uint8_t parameterAddress, const uint32_t value)

Set a configuration setting to a given value.

Parameters

- **moduleAddress** – Module address. An integer number that represents a group of settings.
- **parameterAddress** – Parameter address. An integer number that specifies a parameter within a parameter module group.
- **value** – New value for the configuration.

Throws
`runtime_error` – Exception is thrown if parameter is not available for the device.

inline bool **setDVSBiasSensitivity**(const *BiasSensitivity* sensitivity)

Set DVS chip bias sensitivity preset.

Parameters
sensitivity – DVS sensitivity preset.

Returns
True if configuration was successful, false otherwise.

inline bool **setDVSGlobalHold**(const bool state)

Enable or disable DVXplorer global hold setting.

Parameters
state – True to enable global hold, false to disable.

Returns
True if configuration was successful, false otherwise.

inline bool **setDVXplorerGlobalReset**(const bool state)

Enable or disable DVXplorer global reset setting.

Parameters
state – True to enable global reset, false to disable.

Returns
True if configuration was successful, false otherwise.

inline bool **setDavisReadoutMode**(const *DavisReadoutMode* mode)

Set davis data readout mode. The configuration will be performed if the connected camera is a DAVIS camera.

Parameters
mode – New readout mode

Returns
True if configuration was successful, false otherwise.

inline bool **setDavisColorMode**(const *DavisColorMode* colorMode)

Set davis color mode. The configuration will be performed if the connected camera is a DAVIS camera.

Parameters
colorMode – Color mode, either grayscale or color (if supported).

Returns
True if configuration was successful, false otherwise.

inline bool **setDVXplorerEFPS**(const *DVXeFPS* eFPS)

Set DVXplorer event FPS value. The configuration will be performed if the connected camera is a DVXplorer camera.

Parameters
eFPS – number of event frames per second in readout (if supported).

Returns
True if configuration was successful, false otherwise.

inline *DataReadVariant* **readNext**()

Read a packet from the camera and return a variant of any packet. You can use std::visit with *dv::io::DataReadHandler* to handle each type of packet using callback methods. This method might not maintain timestamp monotonicity between different stream types.

Returns
A variant containing data packet from the camera.

inline bool **handleNext**(*DataReadHandler* &handler)

Read next packet from the camera and use a handler object to handle all types of packets. The function returns a true if end-of-file was not reached, so this function call can be used in a while loop like so:

```
while (camera.handleNext(handler)) {
        // While-loop executes after each packet
}
```

Parameters
handler – Handler instance that contains callback functions to handle different packets.

Returns
False to indicate end of data stream, true to continue.

inline bool **isConnected**() const

Check whether camera is still connected.

Deprecated:

Please use *isRunning()* method instead.

Returns
False if camera is disconnected, true if it is still connected and running.

inline virtual bool **isRunning**() const override

Check whether camera is connected and active.

Returns
True if it is still connected and running, false if camera is disconnected.

inline bool **isMasterCamera**() const

Checks whether the camera is a master camera in multiple camera setups. If camera does not have synchronization cable connected, it will identified as master camera.

Returns
True if camera is master camera, false otherwise.

inline float **getImuRate**() const

Get the configured *IMU* measurement rate. DVXplorer cameras support individual rates for accelerometer and gyroscope, in the case camera configured to have different rates, this function return the lowest value.

Returns
IMU rate in Hz.

inline *std*::string **getImuName**() const

Get *IMU* production model name.

Returns
String containing production model name of the camera on-board *IMU*.

inline *std*::optional<float> **getPixelPitch**() const noexcept

Return pixel pitch distance for the connected camera model. The value is returned in meters, it is:

- DVXplorer Lite - 18 micrometers (1.8e-5)
- DVXplorer and DVXplorer Mini - 9 micrometers (9e-6)
- DAVIS346 and DAVIS240 - 18.5 micrometers (1.85e-5)

Returns
Pixel pitch distance in meters according to the connected device, returns `std::nullopt` if device can't be reliably identified.

inline int64_t **getTimestampOffset**() const

Get the timestamp offset.

Returns
Absolute timestamp offset value.

inline void **setTimestampOffset**(const int64_t timestampOffset)

Set a new timestamp offset value for the camera. This will cause to drop any buffered data captured before calling this method.

Parameters
timestampOffset – New timestamp offset value in microseconds.

inline int64_t **getEventSeekTime**() const

Get latest timestamp of event data stream that has been read from the capture class.

Returns
Latest processed event timestamp; returns -1 if no data was processed or stream is unavailable.

inline int64_t **getFrameSeekTime**() const

Get latest timestamp of frames stream that has been read from the capture class.

Returns
Latest processed frame timestamp; returns -1 if no data was processed or stream is unavailable.

inline int64_t **getImuSeekTime**() const

Get latest timestamp of imu data that has been read from the capture class.

Returns
Latest processed imu data timestamp; returns -1 if no data was processed or stream is unavailable.

inline int64_t **getTriggerSeekTime**() const

Get latest timestamp of trigger data stream that has been read from the capture class.

Returns
Latest processed trigger timestamp; returns -1 if no data was processed or stream is unavailable.

Private Types

enum class **InitialState**

Values:

enumerator **DISCARD_DATA**

enumerator **WAIT_FOR_RESET**

enumerator **DO_MANUAL_RESET**

enumerator **RUNNING**

using **EventPacketPair** = *std*::pair<size_t, *std*::shared_ptr<libcaer::events::EventPacket>>

Private Functions

inline void **discoverMatchingCamera**(const *std*::string &cameraName, const *CameraType* type)

inline void **sendTimestampReset**()

inline bool **isDeviceDVXplorerMini**() const

Checks whether connected device is a DVXplorer Mini model.

Returns

True if the device is a mini, false otherwise

inline explicit **CameraCapture**(const *std*::string &cameraName, const *CameraType* type, const bool doTimestampReset)

Create a camera capture class which opens a camera according to given parameters.

Parameters

- **cameraName** – Camera name, an empty string will match any name.
- **type** – Type of camera, one of: any, DVS, or DAVIS.
- **doTimestampReset** – Reset this camera's timestamps on startup. Required for stereo capture.

Private Members

std::atomic<bool> **keepRunning** = {true}

std::atomic<*InitialState*> **initState** = {*InitialState*::*DISCARD_DATA*}

std::atomic<int64_t> **mTimestampOffset** = {-1}

caer_device_discovery_result **discoveryResult** = {}

std::unique_ptr<libcaer::devices::device> **device** = {nullptr}

SortedPacketBuffers **buffers**

Private Static Functions

static inline float **boschAccRateToFreq**(const uint32_t value)

static inline float **boschGyroRateToFreq**(const uint32_t value)

static inline *dv*::*Frame* **convertFramePacket**(const *std*::shared_ptr<libcaer::events::EventPacket> &packet, const int64_t timestampOffset)

static inline *dv*::*cvector*<*dv*::*IMU*> **convertImuPacket**(const *std*::shared_ptr<libcaer::events::EventPacket> &packet, const int64_t timestampOffset)

static inline *dv*::*cvector*<*dv*::*Trigger*> **convertTriggerPacket**(const *std*::shared_ptr<libcaer::events::EventPacket> &packet, const int64_t timestampOffset, int64_t &maxTimestamp)

static inline *dv*::*EventStore* **convertEventsPacket**(const *std*::shared_ptr<libcaer::events::EventPacket> &packet, const int64_t timestampOffset)

static inline bool **containsResetEvent**(const *std*::shared_ptr<libcaer::events::EventPacket> &packet)

Friends

friend class dv::io::StereoCapture

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *DVXeFPS* &var)

class **CameraGeometry**

Public Types

enum class **FunctionImplementation**

Values:

enumerator **LUT**

enumerator **SubPixel**

using **SharedPtr** = *std*::shared_ptr<*CameraGeometry*>

using **UniquePtr** = *std*::unique_ptr<*CameraGeometry*>

Public Functions

inline **CameraGeometry**(const *std*::vector<float> &distortion, const float fx, const float fy, const float cx, const float cy, const cv::Size &resolution, const *DistortionModel* distortionModel)

Create a camera geometry model with distortion model. Currently only radial tangential model is supported.

Parameters

- **distortion** – Distortion coefficient (4 or 5 coefficient radtan model).
- **fx** – Focal length X measured in pixels.
- **fy** – Focal length Y measured in pixels.
- **cx** – Central point coordinate X in pixels.
- **cy** – Central point coordinate Y in pixels.
- **resolution** – Sensor resolution.

inline **CameraGeometry**(const float fx, const float fy, const float cx, const float cy, const cv::Size &resolution)

Create a camera geometry model without distortion model. Currently only radial tangential model is supported.

Any calls to function dependent on distortion will cause exceptions or segfaults.

Parameters

- **fx** – Focal length X measured in pixels.
- **fy** – Focal length Y measured in pixels.
- **cx** – Central point coordinate X in pixels.
- **cy** – Central point coordinate Y in pixels.
- **resolution** – Sensor resolution.

template<*concepts*::*Coordinate2DCostructible* **Output**, *concepts*::*Coordinate2D* **Input**>
inline *Output* **undistort**(const *Input* &point) const

Returns pixel coordinates of given point with applied back projection, undistortion, and projection. This function uses look-up table and is designed for minimal execution speed.

WARNING: will cause a segfault if coordinates are out-of-bounds or if distortion model is not available.

Parameters
point – Pixel coordinate

Returns
Undistorted pixel coordinate

inline *dv*::*EventStore* **undistortEvents**(const *dv*::*EventStore* &events) const

Undistort event coordinates, discards events which fall beyond camera resolution.

Parameters
events – Input events

Returns
A new event store containing the same events with undistorted coordinates

template<*concepts*::*Coordinate2DMutableIterable* **Output**, *concepts*::*Coordinate2DIterable* **Input**>

inline *Output* **undistortSequence**(const *Input* &coordinates) const

Undistort point coordinates.

Parameters
coordinates – Input point coordinates

Returns
A new vector containing the points with undistorted coordinates

template<*concepts*::*Coordinate3DCostructible* **Output**, *concepts*::*Coordinate3D* **Input**>
inline *Output* **distort**(const *Input* &undistortedPoint) const

Apply distortion to a 3D point.

Parameters
point – Point in 3D space

Returns
Distorted point

template<*concepts*::*Coordinate3DMutableIterable* **Output**, *concepts*::*Coordinate3DIterable* **Input**>
inline *Output* **distortSequence**(const *Input* &points) const

Apply direct distortion on the 3D points.

Parameters
points – Input points

Returns
Distorted points

template<*concepts*::*Coordinate3DCostructible* **Output**, *concepts*::*Coordinate2D* **Input**, *FunctionImplementation* **implementation** = *FunctionImplementation*::*LUT*>
inline *Output* **backProject**(const *Input* &pixel) const

Back-project pixel coordinates into a unit ray vector of depth = 1.0 meters.

Parameters
pixel – Pixel to be projected

Template Parameters
implementation – Specify the internal implementation to performthe computations, `SubPixel` performs all computations without any optimization, `LUT` option avoids computation by perfoming a look-up table operation instead, but rounds input coordinate values.

Returns
Back projected unit ray

template<*concepts*::*Coordinate3DMutableIterable* **Output**, *concepts*::*Coordinate2DIterable* **Input**, *FunctionImplementation* **implementation** = *FunctionImplementation*::*LUT*>
inline *Output* **backProjectSequence**(const *Input* &points) const

Back project a sequence of 2D point into 3D unit ray-vectors.

Parameters
points – Input points.

Template Parameters
implementation – Specify the internal implementation to performthe computations, `SubPixel` performs all computations without any optimization, `LUT` option avoids computation by perfoming a look-up table operation instead, but rounds input coordinate values.

Returns
A sequence of back-projected unit ray vectors.

template<*concepts*::*Coordinate3DCostructible* **Output**, *concepts*::*Coordinate2D* **Input**>
inline *Output* **backProjectUndistort**(const *Input* &pixel) const

Returns a unit ray of given coordinates with applied back projection and undistortion. This function uses look-up table and is designed for minimal execution speed.

WARNING: will cause a segfault if coordinates are out-of-bounds or if distortion model is not available.

Parameters
pixel – Pixel coordinate

Returns
Back projected and undistorted unit ray

template<*concepts*::*Coordinate3DMutableIterable* **Output**, *concepts*::*Coordinate2DIterable* **Input**>
inline *Output* **backProjectUndistortSequence**(const *Input* &points) const

Undistort and back project a batch of points. Output is normalized point coordinates as unit rays.

Parameters
points – Input points.

Returns
Undistorted and back projected points.

template<*concepts*::*Coordinate2DCostructible* **Output**, *concepts*::*Coordinate3D* **Input**>
inline *Output* **project**(const *Input* &points) const

Project a 3D point into pixel plane.

WARNING: Does not perform range checking!

Parameters
points – 3D points to be projected

Returns
Projected pixel coordinates

template<*concepts*::*Coordinate2DMutableIterable* **Output**, *concepts*::*Coordinate3DIterable* **Input**>
inline *Output* **projectSequence**(const *Input* &points, const bool dimensionCheck = true) const

Project a batch of 3D points into pixel plane.

Parameters

- **points** – Points to be projected.
- **dimensionCheck** – Whether to perform resolution check, if true, output points outside of valid frame resolution will be omitted. If disabled, output point count and order will be the same as input points.

Returns
Projected points in pixel plane.

template<*concepts*::*Coordinate2D* **Input**>
inline bool **isWithinDimensions**(const *Input* &point) const

Check whether given coordinates are within valid range.

Parameters
point – Pixel coordinates

Returns
True if the coordinate values are within camera resolution, false otherwise.

inline bool **isUndistortionAvailable**() const

Checks whether this camera geometry calibration contains coefficient for an undistortion model.

Returns
True if undistortion is available, false otherwise

inline cv::Matx33f **getCameraMatrix**() const

Get camera matrix in the format: | mFx 0 mCx | | 0 mFy mCy | | 0 0 1 |

Returns
3x3 Camera matrix with pixel length values

template<*concepts*::*Coordinate2DCostructible* **Output** = cv::Point2f>
inline *Output* **getFocalLength**() const

Focal length

Returns
Focal length in pixels

template<*concepts*::*Coordinate2DCostructible* **Output** = cv::Point2f>
inline *Output* **getCentralPoint**() const

Central point coordinates

Returns
Central point coordinates in pixels

inline *std*::vector<float> **getDistortion**() const

Get distortion coefficients

Returns
Vector containing distortion coefficients

inline *DistortionModel* **getDistortionModel**() const

Get distortion model

Returns
DistortionModel type

inline cv::Size **getResolution**() const

Get the camera resolution.

Returns
Camera sensor resolution

Private Functions

inline void **generateLUTs**()

Generates internal distortion look-up table to speed up undistortion.

template<*concepts*::*Coordinate3DCostructible* **Output**, *concepts*::*Coordinate3D* **Input**>
inline *Output* **distortRadialTangential**(const *Input* &point) const

Distort the Input point according to the Radial Tangential distortion model.

Template Parameters

- **Output** –
- **Input** –

Parameters
point –

Returns
the distorted point in the 3D space

template<*concepts*::*Coordinate3DCostructible* **Output**, *concepts*::*Coordinate3D* **Input**>
inline *Output* **distortEquidistant**(const *Input* &point) const

Distort the Input point according to the Equidistant distortion model.

Template Parameters

- **Output** –
- **Input** –

Parameters
point –

Returns
the distorted point in the 3D space

Private Members

std::vector<cv::Point3f> **mDistortionLUT**

Row-based distortion look-up table. Access index by: index = (y * width) + x

std::vector<cv::Point3f> **mBackProjectLUT**

Row-based distortion look-up table. Access index by: index = (y * width) + x

std::vector<cv::Point2f> **mDistortionPixelLUT**

Row-based undistorted coordinate look-up table, containing undistorted points in pixel space. Access index by: index = (y * width) + x

std::vector<float> **mDistortion**

Distortion coefficients

float **mFx**

Focal length on x axis in pixels

float **mFy**

Focal length on y axis in pixels

float **mCx**

Central point coordinates on x axis

float **mCy**

Central point coordinates on x axis

cv::Size **mResolution**

Sensor resolution

float **mMaxX**

Max floating point coordinate x address value

float **mMaxY**

Max floating point coordinate y address value

DistortionModel **mDistortionModel**

Distortion model used

class **CameraInputBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/camera_input_base.hpp> Camera input base class to abstract live camera and recorded files with a common interface.

Subclassed by *dv::io::CameraCapture*, *dv::io::MonoCameraRecording*, *dv::io::NetworkReader*

Public Functions

virtual *std*::optional<*dv*::*EventStore*> **getNextEventBatch**() = 0

Parse and retrieve next event batch.

Returns
Event batch or `std::nullopt` if no events were received since last read.

virtual *std*::optional<*dv*::*Frame*> **getNextFrame**() = 0

Parse and retrieve next frame.

Returns
Frame or `std::nullopt` if no frames were received since last read.

virtual *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **getNextImuBatch**() = 0

Parse and retrieve next *IMU* data batch.

Returns
IMU data batch or `std::nullopt` if no *IMU* data was received since last read.

virtual *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **getNextTriggerBatch**() = 0

Parse and retrieve next trigger data batch.

Returns
Trigger data batch or `std::nullopt` if no triggers were received since last read.

virtual *std*::optional<cv::Size> **getEventResolution**() const = 0

Get event stream resolution.

Returns
Event stream resolution, `std::nullopt` if event stream is unavailable.

virtual *std*::optional<cv::Size> **getFrameResolution**() const = 0

Retrieve frame stream resolution.

Returns
Frame stream resolution or `std::nullopt` if the frame stream is not available.

virtual bool **isEventStreamAvailable**() const = 0

Check whether event stream is available.

Returns
True if event stream is available, false otherwise.

virtual bool **isFrameStreamAvailable**() const = 0

Check whether frame stream is available.

Returns
True if frame stream is available, false otherwise.

virtual bool **isImuStreamAvailable**() const = 0

Check whether *IMU* data is available.

Returns
True if *IMU* data stream is available, false otherwise.

virtual bool **isTriggerStreamAvailable**() const = 0

Check whether trigger data is available.

Returns
True if trigger data stream is available, false otherwise.

virtual *std*::string **getCameraName**() const = 0

Get camera name, which is a combination of the camera model and the serial number.

Returns
String containing the camera model and serial number separated by an underscore character.

virtual bool **isRunning**() const = 0

Check whether input data streams are still available. For a live camera this should check whether device is still connected and functioning, while for a recording file this should check whether end of stream was reached using sequential reads.

Returns
True if data read is possible, false otherwise.

class **CameraOutputBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/camera_output_base.hpp> Output reader base class defining API interface for writing camera data into an IO resource.

Subclassed by *dv::io::NetworkWriter*

Public Functions

virtual void **writeEvents**(const *dv*::*EventStore* &events) = 0

Write event data into the output.

Parameters
events – Write events into the output.

virtual void **writeFrame**(const *dv*::*Frame* &frame) = 0

Write a frame into the output.

Parameters
frame – Write a frame into the output.

virtual void **writeIMU**(const *dv*::*cvector*<*dv*::*IMU*> &imu) = 0

Write imu data into the output.

Parameters
imu – Write imu into the output.

virtual void **writeTriggers**(const *dv*::*cvector*<*dv*::*Trigger*> &triggers) = 0

Write trigger data into the output.

Parameters
triggers – Write trigger into the output.

virtual *std*::string **getCameraName**() const = 0

Retrieve camera name of this writer output instance.

Returns
Configured camera name.

template<size_t **radius**>

struct **CircleCoordinates**

template<>

struct **CircleCoordinates**<3>

Public Static Attributes

static *std*::vector<Eigen::Vector2i, Eigen::aligned_allocator<Eigen::Vector2i>> **coords**{{Eigen::Vector2i{0, 3}, Eigen::Vector2i{1, 3}, Eigen::Vector2i{2, 2}, Eigen::Vector2i{3, 1}, Eigen::Vector2i{3, 0}, Eigen::Vector2i{3, -1}, Eigen::Vector2i{2, -2}, Eigen::Vector2i{1, -3}, Eigen::Vector2i{0, -3}, Eigen::Vector2i{-1, -3}, Eigen::Vector2i{-2, -2}, Eigen::Vector2i{-3, -1}, Eigen::Vector2i{-3, 0}, Eigen::Vector2i{-3, 1}, Eigen::Vector2i{-2, 2}, Eigen::Vector2i{-1, 3}}}

template<>

struct **CircleCoordinates**<4>

Public Static Attributes

static *std*::vector<Eigen::Vector2i, Eigen::aligned_allocator<Eigen::Vector2i>> **coords**{{Eigen::Vector2i{0, 4}, Eigen::Vector2i{1, 4}, Eigen::Vector2i{2, 3}, Eigen::Vector2i{3, 2}, Eigen::Vector2i{4, 1}, Eigen::Vector2i{4, 0}, Eigen::Vector2i{4, -1}, Eigen::Vector2i{3, -2}, Eigen::Vector2i{2, -3}, Eigen::Vector2i{1, -4}, Eigen::Vector2i{0, -4}, Eigen::Vector2i{-1, -4}, Eigen::Vector2i{-2, -3}, Eigen::Vector2i{-3, -2}, Eigen::Vector2i{-4, -1}, Eigen::Vector2i{-4, 0}, Eigen::Vector2i{-4, 1}, Eigen::Vector2i{-3, 2}, Eigen::Vector2i{-2, 3}, Eigen::Vector2i{-1, 4}}}

template<>

struct **CircleCoordinates**<5>

Public Static Attributes

static *std*::vector<Eigen::Vector2i, Eigen::aligned_allocator<Eigen::Vector2i>> **coords**{{Eigen::Vector2i{0, 5}, Eigen::Vector2i{1, 5}, Eigen::Vector2i{2, 5}, Eigen::Vector2i{3, 4}, Eigen::Vector2i{4, 3}, Eigen::Vector2i{5, 2}, Eigen::Vector2i{5, 1}, Eigen::Vector2i{5, 0}, Eigen::Vector2i{5, -1}, Eigen::Vector2i{5, -2}, Eigen::Vector2i{4, -3}, Eigen::Vector2i{3, -4}, Eigen::Vector2i{2, -5}, Eigen::Vector2i{1, -5}, Eigen::Vector2i{0, -5}, Eigen::Vector2i{-1, -5}, Eigen::Vector2i{-2, -5}, Eigen::Vector2i{-3, -4}, Eigen::Vector2i{-4, -3}, Eigen::Vector2i{-5, -2}, Eigen::Vector2i{-5, -1}, Eigen::Vector2i{-5, 0}, Eigen::Vector2i{-5, 1}, Eigen::Vector2i{-5, 2}, Eigen::Vector2i{-4, 3}, Eigen::Vector2i{-3, 4}, Eigen::Vector2i{-2, 5}, Eigen::Vector2i{-1, 5}}}

template<>

struct **CircleCoordinates**<6>

Public Static Attributes

static *std*::vector<Eigen::Vector2i, Eigen::aligned_allocator<Eigen::Vector2i>> **coords**{{Eigen::Vector2i{0, 6}, Eigen::Vector2i{1, 6}, Eigen::Vector2i{2, 6}, Eigen::Vector2i{3, 5}, Eigen::Vector2i{4, 4}, Eigen::Vector2i{5, 3}, Eigen::Vector2i{6, 2}, Eigen::Vector2i{6, 1}, Eigen::Vector2i{6, 0}, Eigen::Vector2i{6, -1}, Eigen::Vector2i{6, -2}, Eigen::Vector2i{5, -3}, Eigen::Vector2i{4, -4}, Eigen::Vector2i{3, -5}, Eigen::Vector2i{2, -6}, Eigen::Vector2i{1, -6}, Eigen::Vector2i{0, -6}, Eigen::Vector2i{-1, -6}, Eigen::Vector2i{-2, -6}, Eigen::Vector2i{-3, -5}, Eigen::Vector2i{-4, -4}, Eigen::Vector2i{-5, -3}, Eigen::Vector2i{-6, -2}, Eigen::Vector2i{-6, -1}, Eigen::Vector2i{-6, 0}, Eigen::Vector2i{-6, 1}, Eigen::Vector2i{-6, 2}, Eigen::Vector2i{-5, 3}, Eigen::Vector2i{-4, 4}, Eigen::Vector2i{-3, 5}, Eigen::Vector2i{-2, 6}, Eigen::Vector2i{-1, 6}}}

template<>

struct **CircleCoordinates**<7>

Public Static Attributes

static *std*::vector<Eigen::Vector2i, Eigen::aligned_allocator<Eigen::Vector2i>> **coords**{{Eigen::Vector2i{0, 7}, Eigen::Vector2i{1, 7}, Eigen::Vector2i{2, 7}, Eigen::Vector2i{3, 7}, Eigen::Vector2i{4, 6}, Eigen::Vector2i{5, 5}, Eigen::Vector2i{6, 4}, Eigen::Vector2i{7, 3}, Eigen::Vector2i{7, 2}, Eigen::Vector2i{7, 1}, Eigen::Vector2i{7, 0}, Eigen::Vector2i{7, -1}, Eigen::Vector2i{7, -2}, Eigen::Vector2i{7, -3}, Eigen::Vector2i{6, -4}, Eigen::Vector2i{5, -5}, Eigen::Vector2i{4, -6}, Eigen::Vector2i{3, -7}, Eigen::Vector2i{2, -7}, Eigen::Vector2i{1, -7}, Eigen::Vector2i{0, -7}, Eigen::Vector2i{-1, -7}, Eigen::Vector2i{-2, -7}, Eigen::Vector2i{-3, -7}, Eigen::Vector2i{-4, -6}, Eigen::Vector2i{-5, -5}, Eigen::Vector2i{-6, -4}, Eigen::Vector2i{-7, -3}, Eigen::Vector2i{-7, -2}, Eigen::Vector2i{-7, -1}, Eigen::Vector2i{-7, 0}, Eigen::Vector2i{-7, 1}, Eigen::Vector2i{-7, 2}, Eigen::Vector2i{-7, 3}, Eigen::Vector2i{-6, 4}, Eigen::Vector2i{-5, 5}, Eigen::Vector2i{-4, 6}, Eigen::Vector2i{-3, 7}, Eigen::Vector2i{-2, 7}, Eigen::Vector2i{-1, 7}, Eigen::Vector2i{0, 7}}}

class **CircularTimeSurfaceView**

Public Types

using **CoordVector** = *std*::vector<Eigen::Vector2i, Eigen::aligned_allocator<Eigen::Vector2i>>

Public Functions

inline explicit **CircularTimeSurfaceView**(*CoordVector* &coords)

inline explicit **CircularTimeSurfaceView**(*CoordVector* &&coords)

inline auto **getTimestamp**(const *dv*::Event &e, const Eigen::Vector2i &circleCoords, const *TimeSurface* &ts) const

template<typename **ITERATOR**>
inline auto **circularIncrement**(const *ITERATOR* it) const

template<typename **ITERATOR**>
inline auto **circularDecrement**(const *ITERATOR* it) const

Public Members

CoordVector **mCoords**

class **CompressionSupport**

Subclassed by *dv::io::compression::Lz4CompressionSupport*, *dv::io::compression::NoneCompressionSupport*, *dv::io::compression::ZstdCompressionSupport*

Public Functions

inline explicit **CompressionSupport**(const *CompressionType* type)

virtual **~CompressionSupport**() = default

virtual void **compress**(*dv*::*io*::*support*::*IODataBuffer* &packet) = 0

inline *CompressionType* **getCompressionType**() const

Private Members

CompressionType **mType**

class **Config**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/mono_camera_writer.hpp> A configuration structure for the *MonoCameraWriter*.

Public Functions

inline void **addStreamMetadata**(const *std*::string &name, const *std*::pair<*std*::string, *dv*::*io*::*support*::*VariantValueOwning*> &metadataEntry)

Add a metadata entry for a data type stream.

Parameters

- **name** – Name of the stream.
- **metadataEntry** – Metadata entry consisting of a pair, where first element is the key name of the stream and second element is the value.

inline void **addEventStream**(const cv::Size &resolution, const *std*::string &name = "events", const *std*::optional<*std*::string> &source = *std*::nullopt)

Add an event stream with a given resolution.

Parameters

- **resolution** – Resolution of the event sensor.
- **name** – Name of the stream
- **source** – Name of the source camera.

inline void **addFrameStream**(const cv::Size &resolution, const *std*::string &name = "frames", const *std*::optional<*std*::string> &source = *std*::nullopt)

Add a frame stream with a given resolution.

Parameters

- **resolution** – Resolution of the frame sensor.
- **name** – Name of the stream
- **source** – Name of the source camera.

inline void **addImuStream**(const *std*::string &name = "imu", const *std*::optional<*std*::string> &source = *std*::nullopt)

Add an imu data stream.

Parameters

name – *Stream* name, with a default value of “imu”.

inline void **addTriggerStream**(const *std*::string &name = "triggers", const *std*::optional<*std*::string> &source = *std*::nullopt)

Add a trigger stream.

Parameters

name – *Stream* name, with a default value of “triggers”.

template<class **PacketType**>
inline void **addStream**(const *std*::string &name, const *std*::optional<*std*::string> &source = *std*::nullopt)

Add a stream of given data type.

Template Parameters

PacketType – *Stream* data packet type.

Parameters

- **name** – Name for the stream.

- **source** – Camera name for the source of the data, usually a concatenation of "MODEL_SERIAL", e.g. "DVXplorer_DXA000000"

inline *std*::optional<cv::Size> **findStreamResolution**(const *std*::string &name) const

Parse resolution of the stream from metadata of the stream. Resolution should be set as two metadata parameters: "sizeX" and "sizeY" parameters.

Parameters
name – *Stream* name.

Returns
Configured resolution. `std::nullopt` if unavailable or incorrectly configured.

inline explicit **Config**(const *std*::string &cameraName, *CompressionType* compression = *CompressionType*::*LZ4*)

Create a config instance

Parameters

- **cameraName** –
- **compression** –

Public Members

dv::*CompressionType* **compression**

Compression type for this file.

std::string **cameraName**

Camera name that produces the data, usually contains production serial number.

Private Members

std::map<*std*::string, *std*::string> **customDataStreams**

std::map<*std*::string, *std*::map<*std*::string, *dv*::*io*::*support*::*VariantValueOwning*>> **customDataStreamsMetadata**

Friends

friend class dv::io::MonoCameraWriter

friend class dv::io::StereoCameraWriter

class **Connection** : public *std*::enable_shared_from_this<*Connection*>

Connection helper class that maintains shared pointer to itself when called on the public API methods.

This class should be wrapped in a shared pointer and start method should be called. This will intrinsically increment the reference count to maintain the pointer to itself even if the wrapper shared_ptr goes out-of-scope until the instance gets API calls to write data into the buffer. During destruction, the instance will remove it's own pointer from a connection list in the top-level class.

(Personal comment by Rokas): this seems over-engineered and unnecessary, but it's the way ASIO works and, although there are other ways to implement it, it just doesn't work with other approaches leading to undefined behaviors.

Public Functions

inline **Connection**(WriteOrderedSocket &&socket, *NetworkWriter* *const server)

inline **~Connection**()

inline void **start**()

inline void **close**()

inline void **writePacket**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> &packet)

inline bool **isOpen**() const

Private Functions

inline void **writeIOHeader**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> &ioHeader)

inline void **keepAliveByReading**()

inline void **handleError**(const boost::system::error_code &error, const *std*::string_view message)

Private Members

NetworkWriter ***mParent**

WriteOrderedSocket **mSocket**

uint8_t **mKeepAliveReadSpace** = {0}

template<class **Functor**>

class **ContrastMaximizationWrapper**

#include </builds/inivation/dv/dv-processing/include/dv-processing/optimization/contrast_maximization_wrapper.hpp>
Wrapper for all contrast maximization algorithms. For more information about contrast maximization please check "contrast_maximization_rotation.hpp" or "contrast_maximization_translation_and_depth.hpp". This wrapper is mainly meant to set the non linear differentiation parameters (see contructor for more information). In addition, the class expose to user only "optimize" function which returns a struct containing the result of the non-linear optimization (successful or not), number of iteration of the optimization and optimized parameters.

Template Parameters

Functor – Functor that handles optimization. Cost is computed by overriding operator() method. For an example of a functor please check "contrast_maximization_rotation.hpp" or "contrast_maximization_translation_and_depth.hpp".

Public Functions

inline **ContrastMaximizationWrapper**(*std*::unique_ptr<*Functor*> functor_, float learningRate, float epsfcn = 0, float ftol = 0.000345267, float gtol = 0, float xtol = 0.000345267, int maxfev = 400)

Parameters

- **functor_** – functor handling contrast maximization optimization. the functor should inherit "OptimizationFunctor" and overload the "int operator()" method to compute cost for contrast maximization and optimize pre-defined parameters.
- **learningRate** – constant multiplying input value to find new value at which function will be evaluated. E.g. assuming function is evaluated at x –> f(x), next input sample x' is computed as x' = abs(x) * learningRate.
- **epsfcn** – error precision
- **ftol** – tolerance for the norm of the vector function
- **gtol** – tolerance for the norm of the gradient of the error vector
- **xtol** – tolerance for the norm of the solution vector
- **maxfev** – max number of function evaluations Note that default parameters are taken from default parameters of LevenbergMarquardt optimizer.

inline optimizationOutput **optimize**(const Eigen::VectorXf &initialValues)

Function optimizing cost defined in mFunctor (inside operator() method).

Parameters
initialValues – Initial values of variables to be optimized.

Returns
optimized variable that minimize cost.

Private Members

std::unique_ptr<*Functor*> **mFunctor** = nullptr

optimizationParameters **mParams**

template<class **T**>

class **cPtrIterator**

Public Types

using **iterator_category** = *std*::random_access_iterator_tag

using **value_type** = typename *std*::remove_cv_t<*T*>

using **pointer** = *T**

using **reference** = *T*&

using **size_type** = size_t

using **difference_type** = ptrdiff_t

Public Functions

constexpr **cPtrIterator**() noexcept = default

inline constexpr **cPtrIterator**(*pointer* elementPtr) noexcept

inline constexpr *reference* **operator***() const noexcept

inline constexpr *pointer* **operator->**() const noexcept

inline constexpr *reference* **operator[]**(const *size_type* index) const noexcept

inline constexpr bool **operator==**(const *cPtrIterator* &rhs) const noexcept

inline constexpr bool **operator!=**(const *cPtrIterator* &rhs) const noexcept

inline constexpr bool **operator<**(const *cPtrIterator* &rhs) const noexcept

inline constexpr bool **operator>**(const *cPtrIterator* &rhs) const noexcept

inline constexpr bool **operator<=**(const *cPtrIterator* &rhs) const noexcept

inline constexpr bool **operator>=**(const *cPtrIterator* &rhs) const noexcept

inline *cPtrIterator* &**operator++**() noexcept

inline *cPtrIterator* **operator++**(int) noexcept

inline *cPtrIterator* &**operator--**() noexcept

inline *cPtrIterator* **operator--**(int) noexcept

inline *cPtrIterator* &**operator+=**(const *size_type* add) noexcept

inline constexpr *cPtrIterator* **operator+**(const *size_type* add) const noexcept

inline *cPtrIterator* &**operator-=**(const *size_type* sub) noexcept

inline constexpr *cPtrIterator* **operator-**(const *size_type* sub) const noexcept

inline constexpr *difference_type* **operator-**(const *cPtrIterator* &rhs) const noexcept

inline void **swap**(*cPtrIterator* &rhs) noexcept

inline constexpr **operator** ***cPtrIterator*<const *value_type*>**() const noexcept

Private Members

pointer **mElementPtr** = {nullptr}

Friends

inline friend constexpr *cPtrIterator* **operator+**(const *size_type* lhs, const *cPtrIterator* &rhs) noexcept

template<class **T**>

class **cvector**

Public Types

using **value_type** = *T*

using **const_value_type** = const *T*

using **pointer** = *T**

using **const_pointer** = const *T**

using **reference** = *T*&

using **const_reference** = const *T*&

using **size_type** = size_t

using **difference_type** = ptrdiff_t

using **iterator** = *cPtrIterator*<*value_type*>

using **const_iterator** = *cPtrIterator*<*const_value_type*>

using **reverse_iterator** = *std*::reverse_iterator<*iterator*>

using **const_reverse_iterator** = *std*::reverse_iterator<*const_iterator*>

Public Functions

constexpr **cvector**() noexcept = default

inline **~cvector**() noexcept

inline **cvector**(const *cvector* &vec, const *size_type* pos = 0, const *size_type* count = *npos*)

template<typename **U**>
inline **cvector**(const *U* &vec, const *size_type* pos = 0, const *size_type* count = *npos*)

inline **cvector**(*const_pointer* vec, const *size_type* vecLength, const *size_type* pos = 0, const *size_type* count = *npos*)

inline explicit **cvector**(const *size_type* count)

inline **cvector**(const *size_type* count, *const_reference* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline **cvector**(*InputIt* first, *InputIt* last)

inline **cvector**(*std*::initializer_list<*value_type*> init_list)

inline **cvector**(*cvector* &&rhs) noexcept

inline *cvector* &**operator=**(*cvector* &&rhs) noexcept

inline *cvector* &**operator=**(const *cvector* &rhs)

template<typename **U**>
inline *cvector* &**operator=**(const *U* &rhs)

inline *cvector* &**operator=**(*const_reference* value)

inline *cvector* &**operator=**(*std*::initializer_list<*value_type*> rhs_list)

inline bool **operator==**(const *cvector* &rhs) const noexcept

inline auto **operator<=>**(const *cvector* &rhs) const noexcept

template<typename **U**>
inline bool **operator==**(const *U* &rhs) const noexcept

template<typename **U**>
inline auto **operator<=>**(const *U* &rhs) const noexcept

inline *cvector* &**assign**(*cvector* &&vec)

inline *cvector* &**assign**(const *cvector* &vec, const *size_type* pos = 0, const *size_type* count = *npos*)

template<typename **U**>
inline *cvector* &**assign**(const *U* &vec, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *cvector* &**assign**(*const_pointer* vec, const *size_type* vecLength, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *cvector* &**assign**(*const_reference* value)

inline *cvector* &**assign**(const *size_type* count, *const_reference* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline *cvector* &**assign**(*InputIt* first, *InputIt* last)

inline *cvector* &**assign**(*std*::initializer_list<*value_type*> init_list)

inline *pointer* **data**() noexcept

inline *const_pointer* **data**() const noexcept

inline *size_type* **size**() const noexcept

inline *size_type* **capacity**() const noexcept

inline *size_type* **max_size**() const noexcept

inline bool **empty**() const noexcept

inline void **resize**(const *size_type* newSize)

inline void **resize**(const *size_type* newSize, *const_reference* value)

inline void **reserve**(const *size_type* minCapacity)

inline void **shrink_to_fit**()

template<typename **INT**>
inline *reference* **operator[]**(const *INT* index)

template<typename **INT**>
inline *const_reference* **operator[]**(const *INT* index) const

template<typename **INT**>
inline *reference* **at**(const *INT* index)

template<typename **INT**>
inline *const_reference* **at**(const *INT* index) const

inline explicit **operator** ***std*::vector<*value_type*>**() const

inline *reference* **front**()

inline *const_reference* **front**() const

inline *reference* **back**()

inline *const_reference* **back**() const

inline void **push_back**(*const_reference* value)

inline void **push_back**(*value_type* &&value)

template<class ...**Args**>
inline *reference* **emplace_back**(*Args*&&... args)

inline void **pop_back**()

inline void **clear**() noexcept

inline void **swap**(*cvector* &rhs) noexcept

inline *iterator* **begin**() noexcept

inline *iterator* **end**() noexcept

inline *const_iterator* **begin**() const noexcept

inline *const_iterator* **end**() const noexcept

inline *const_iterator* **cbegin**() const noexcept

inline *const_iterator* **cend**() const noexcept

inline *reverse_iterator* **rbegin**() noexcept

inline *reverse_iterator* **rend**() noexcept

inline *const_reverse_iterator* **rbegin**() const noexcept

inline *const_reverse_iterator* **rend**() const noexcept

inline *const_reverse_iterator* **crbegin**() const noexcept

inline *const_reverse_iterator* **crend**() const noexcept

inline *iterator* **insert**(*const_iterator* pos, *const_reference* value)

inline *iterator* **insert**(*const_iterator* pos, *value_type* &&value)

inline *iterator* **insert**(*const_iterator* pos, const *size_type* count, *const_reference* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline *iterator* **insert**(*const_iterator* pos, *InputIt* first, *InputIt* last)

inline *iterator* **insert**(*const_iterator* pos, *std*::initializer_list<*value_type*> init_list)

template<class ...**Args**>
inline *iterator* **emplace**(*const_iterator* pos, *Args*&&... args)

inline *iterator* **erase**(*const_iterator* pos)

inline *iterator* **erase**(*const_iterator* first, *const_iterator* last)

inline *cvector* &**append**(const *cvector* &vec, const *size_type* pos = 0, const *size_type* count = *npos*)

template<typename **U**>
inline *cvector* &**append**(const *U* &vec, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *cvector* &**append**(*const_pointer* vec, const *size_type* vecLength, const *size_type* pos = 0, const *size_type* count = *npos*)

inline *cvector* &**append**(*const_reference* value)

inline *cvector* &**append**(const *size_type* count, *const_reference* value)

template<typename **InputIt**, *std*::enable_if_t<*std*::is_base_of_v<*std*::input_iterator_tag, typename *std*::iterator_traits<*InputIt*>::iterator_category>, bool> = true>
inline *cvector* &**append**(*InputIt* first, *InputIt* last)

inline *cvector* &**append**(*std*::initializer_list<*value_type*> init_list)

inline *cvector* &**operator+=**(const *cvector* &rhs)

template<typename **U**>
inline *cvector* &**operator+=**(const *U* &rhs)

inline *cvector* &**operator+=**(*const_reference* value)

inline *cvector* &**operator+=**(*std*::initializer_list<*value_type*> rhs_list)

inline *cvector* **operator+**(const *cvector* &rhs) const

template<typename **U**>
inline *cvector* **operator+**(const *U* &rhs) const

inline *cvector* **operator+**(*const_reference* value) const

inline *cvector* **operator+**(*std*::initializer_list<*value_type*> rhs_list) const

template<typename **U**>
inline bool **contains**(const *U* &item) const

template<typename **Pred**>
inline bool **containsIf**(*Pred* predicate) const

inline void **sortUnique**()

template<typename **Compare**>
inline void **sortUnique**(*Compare* comp)

template<typename **U**>
inline *size_type* **remove**(const *U* &item)

template<typename **Pred**>
inline *size_type* **removeIf**(*Pred* predicate)

Public Static Attributes

static constexpr *size_type* **npos** = {static_cast<*size_type*>(-1)}

Private Functions

inline void **ensureCapacity**(const *size_type* newSize)

inline void **reallocateMemory**(const *size_type* newSize)

template<bool **CHECKED**>
inline *size_type* **getIndex**(const *size_type* index) const

template<bool **CHECKED**>
inline *size_type* **getIndex**(const *difference_type* index) const

Private Members

size_type **mCurrSize** = {0}

size_type **mMaximumSize** = {0}

pointer **mDataPtr** = {nullptr}

Friends

template<typename **U**>
inline friend *cvector* **operator+**(const *U* &lhs, const *cvector* &rhs)

inline friend *cvector* **operator+**(*const_reference* value, const *cvector* &rhs)

inline friend *cvector* **operator+**(*std*::initializer_list<*value_type*> lhs_list, const *cvector* &rhs)

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *cvector* &rhs)

struct **DataReadHandler**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/data_read_handler.hpp> Read handler that can handle all supported types in *MonoCameraRecording*.

Public Types

enum class **OutputFlag**

Values:

enumerator **EndOfFile**

enumerator **Continue**

Public Functions

inline void **operator()**(const *dv*::*EventStore* &events)

Internal call to handle input data

Parameters
events –

inline void **operator()**(const *dv*::*Frame* &frame)

Internal call to handle input data

Parameters
frame –

inline void **operator()**(const *dv*::*cvector*<*dv*::*Trigger*> &triggers)

Internal call to handle input data

Parameters
triggers –

inline void **operator()**(const *dv*::*cvector*<*dv*::*IMU*> &imu)

Internal call to handle input data

Parameters
imu –

inline void **operator()**(const *OutputFlag* flag)

Internal call to handle input data

Parameters
flag –

Public Members

std::optional<*std*::function<void(const *dv*::*EventStore*&)>> **mEventHandler** = *std*::nullopt

Event handler that is going to be called on each arriving event batch.

std::optional<*std*::function<void(const *dv*::*Frame*&)>> **mFrameHandler** = *std*::nullopt

Frame handler that is called on each arriving frame.

std::optional<*std*::function<void(const *dv*::*cvector*<*dv*::*IMU*>&)>> **mImuHandler** = *std*::nullopt

IMU data handler that is going to be called on each arriving imu data batch.

std::optional<*std*::function<void(const *dv*::*cvector*<*dv*::*Trigger*>&)>> **mTriggersHandler** = *std*::nullopt

Trigger data handler that is going to be called on each arriving trigger data batch.

std::optional<*std*::function<void(const *OutputFlag*)>> **mOutputFlagHandler** = *std*::nullopt

A handler for output flags that can indicate some file behaviour, e.g. end-of-file.

bool **eof** = false

Is end of file reached.

int64_t **seek** = -1

Timestamp holding latest seek position of the recording

class **DecompressionSupport**

Subclassed by *dv::io::compression::Lz4DecompressionSupport*, *dv::io::compression::NoneDecompressionSupport*, *dv::io::compression::ZstdDecompressionSupport*

Public Functions

inline explicit **DecompressionSupport**(const *CompressionType* type)

virtual **~DecompressionSupport**() = default

virtual void **decompress**(*std*::vector<*std*::byte> &source, *std*::vector<*std*::byte> &target) = 0

inline *CompressionType* **getCompressionType**() const

Private Members

CompressionType **mType**

struct **Depth**

#include </builds/inivation/dv/dv-processing/include/dv-processing/measurements/depth.hpp> A depth measurement structure that contains a timestamped measurement of depth.

Public Functions

inline **Depth**(int64_t timestamp, float depth)

Public Members

int64_t **mTimestamp**

UNIX Microsecond timestamp

float **mDepth**

Depth measurement value, expected to be in meters.

struct **DepthEventPacket** : public *flatbuffers*::NativeTable

Public Types

typedef *DepthEventPacketFlatbuffer* **TableType**

Public Functions

inline **DepthEventPacket**()

inline **DepthEventPacket**(const *dv*::*cvector*<DepthEvent> &_elements)

Public Members

dv::*cvector*<DepthEvent> **elements**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *DepthEventPacket* &packet)

struct **DepthEventPacketBuilder**

Public Functions

inline void **add_elements**(*flatbuffers*::Offset<*flatbuffers*::Vector<const DepthEvent*>> elements)

inline explicit **DepthEventPacketBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

DepthEventPacketBuilder &**operator=**(const *DepthEventPacketBuilder*&)

inline *flatbuffers*::Offset<*DepthEventPacketFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **DepthEventPacketFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *DepthEventPacket* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<const DepthEvent*> ***elements**()* const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *DepthEventPacket* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*DepthEventPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*DepthEventPacket* *_o, const *DepthEventPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*DepthEventPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *DepthEventPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "DEVT"

struct **DepthFrame** : public *flatbuffers*::NativeTable

Public Types

typedef *DepthFrameFlatbuffer* **TableType**

Public Functions

inline **DepthFrame**()

inline **DepthFrame**(int64_t _timestamp, int16_t _sizeX, int16_t _sizeY, uint16_t _minDepth, uint16_t _maxDepth, uint16_t _step, const *dv*::*cvector*<uint16_t> &_depth)

Public Members

int64_t **timestamp**

int16_t **sizeX**

int16_t **sizeY**

uint16_t **minDepth**

uint16_t **maxDepth**

uint16_t **step**

dv::*cvector*<uint16_t> **depth**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *DepthFrame* &frame)

struct **DepthFrameBuilder**

Public Functions

inline void **add_timestamp**(int64_t timestamp)

inline void **add_sizeX**(int16_t sizeX)

inline void **add_sizeY**(int16_t sizeY)

inline void **add_minDepth**(uint16_t minDepth)

inline void **add_maxDepth**(uint16_t maxDepth)

inline void **add_step**(uint16_t step)

inline void **add_depth**(*flatbuffers*::Offset<*flatbuffers*::Vector<uint16_t>> depth)

inline explicit **DepthFrameBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

DepthFrameBuilder &**operator=**(const *DepthFrameBuilder*&)

inline *flatbuffers*::Offset<*DepthFrameFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **DepthFrameFlatbuffer** : private *flatbuffers*::Table

#include </builds/inivation/dv/dv-processing/include/dv-processing/data/depth_frame_base.hpp> A frame containing pixel depth values in millimeters.

Public Types

typedef *DepthFrame* **NativeTableType**

Public Functions

inline int64_t **timestamp**() const

Central timestamp (µs), corresponds to exposure midpoint.

inline int16_t **sizeX**() const

Start of *Frame* (SOF) timestamp.

inline int16_t **sizeY**() const

Y axis length in pixels.

inline uint16_t **minDepth**() const

Minimum valid depth value.

inline uint16_t **maxDepth**() const

Maximum valid depth value.

inline uint16_t **step**() const

Depth step value, minimal depth distance that can be measured by the sensor setup.

inline const *flatbuffers*::Vector<uint16_t> ***depth**() const

Depth values, unsigned 16bit integers, millimeters from the camera frame, following the OpenNI standard. Depth value of 0 should be considered an invalid value.

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *DepthFrame* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*DepthFrame* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable***()

static inline constexpr const char ***GetFullyQualifiedName***()

static inline void **UnPackToFrom**(*DepthFrame* *_o, const *DepthFrameFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*DepthFrameFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *DepthFrame* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier*** = "DFRM"

struct **DirectoryError**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **DirectoryNotFound**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

class **EdgeMapAccumulator** : public *dv*::*AccumulatorBase*

#include *</builds/inivation/dv/dv-processing/include/dv-processing/core/frame/edge_map_accumulator.hpp>*
`dv::EdgeMapAccumulator` accumulates events in a histogram representation with configurable contribution, but it is more efficient compared to generic accumulator since it uses 8-bit unsigned integers as internal memory type.

The *EdgeMapAccumulator* behaves the same as a generic `dv::Accumulator` with STEP decay function, neutral and minimum value of 0.0, maximum value of 1.0 and configurable event contribution. The difference is that

it doesn't use floating point numbers for the potential surface representation. The output data type of this accumulator is single channel 8-bit unsigned integer (CV_8UC1). Accumulation is performed using integer operations as well. Due to performance, no check on the event coordinates inside image plane is performed, unless compiled specifically in DEBUG mode. Events out of the image plane bounds will result in undefined behaviour, or program termination in DEBUG mode.

Public Functions

inline explicit **EdgeMapAccumulator**(const cv::Size &resolution, const float contribution_ = 0.25f, const bool ignorePolarity_ = true, const float neutralPotential = 0.f, const float decay_ = *EdgeMapAccumulator*::DECAY_FULL)

Create a pixel accumulator with known image dimensions and event contribution.

Parameters

- **resolution** – Dimensions of the expected event sensor
- **contribution_** – Contribution coefficient for a single event. The contribution value is multiplied by the maximum possible pixel value (255) to get the increment value. E.g. contribution value of 0.1 will increment a pixel value at a single event coordinates by 26.
- **ignorePolarity_** – Set ignore polarity option. All events are considered positive if enabled.
- **neutralPotential** – Neutral potential value. Neutral value is the default pixel value when decay is disabled and the value that pixels decay into when decay is enabled. The range for neutral potential value is [0.0; 1.0], where 1.0 stands for maximum possible potential - 255 in 8-bit pixel representation.
- **decay_** – Decay coefficient value. This value defines how fast pixel values decay to neutral value. The bigger the value the faster the pixel value will reach neutral value. Decay is applied before each frame generation. The range for decay value is [0.0; 1.0], where 0.0 will not apply any decay and 1.0 will apply maximum decay value resetting a pixel to neutral potential at each generation (default behavior).

inline float **getContribution**() const

Get the contribution coefficient for a single event. The contribution value is multiplied by the maximum possible pixel value (255) to get the increment value. E.g. contribution value of 0.1 will increment a pixel value at a single event coordinates by 26.

Deprecated:

Use *getEventContribution()* method instead.

See also:

dv::EdgeMapAccumulator::getEventContribution

Returns

Contribution coefficient

inline void **setContribution**(const float contribution_)

Set new contribution coefficient.

Deprecated:

Use *setEventContribution()* method instead.

See also:

dv::EdgeMapAccumulator::setEventContribution

Parameters
contribution_ – Contribution coefficient for a single event. The contribution value is multiplied by the maximum possible pixel value (255) to get the increment value. E.g. contribution value of 0.1 will increment a pixel value at a single event coordinates by 26.

inline float **getEventContribution**() const

Get the contribution coefficient for a single event. The contribution value is multiplied by the maximum possible pixel value (255) to get the increment value. E.g. contribution value of 0.1 will increment a pixel value at a single event coordinates by 26.

Returns
Contribution coefficient

inline void **setEventContribution**(const float contribution_)

Set new contribution coefficient.

Parameters
contribution_ – Contribution coefficient for a single event. The contribution value is multiplied by the maximum possible pixel value (255) to get the increment value. E.g. contribution value of 0.1 will increment a pixel value at a single event coordinates by 26.

inline virtual void **accumulate**(const *EventStore* &packet) override

Perform accumulation on given events.

Parameters
packet – Event store containing event to be accumulated.

inline virtual *dv*::*Frame* **generateFrame**() override

Generates the accumulation frame (potential surface) at the time of the last consumed event. The function writes the output image into the given `outFrame` argument. The output frame will contain data with type CV_8UC1.

The function resets any events accumulated up to this function call.

Parameters
frame – the frame to generate the image to

inline void **reset**()

Clear the buffered events.

inline *EdgeMapAccumulator* &**operator<<**(const *EventStore* &store)

Accumulates the event store into the accumulator.

Parameters
store – The event store to be accumulated.

Returns
A reference to this *EdgeMapAccumulator*.

inline bool **isIgnorePolarity**() const

Check whether ignore polarity option is set to true.

Returns
True if the accumulator assumes all events as positive, false otherwise.

inline void **setIgnorePolarity**(const bool ignorePolarity_)

Set ignore polarity option. All events are considered positive if enabled.

Parameters
ignorePolarity_ – True to enable ignore polarity option.

inline float **getNeutralValue**() const

Get the neutral potential value for the accumulator. The range for potential value is [0.0; 1.0], where 1.0 stands for maximum possible potential - 255 in 8-bit pixel representation.

Deprecated:

Use *getNeutralPotential()* method instead.

See also:

dv::EdgeMapAccumulator::getNeutralPotential

Returns
Neutral potential value in range [0.0; 1.0]

inline void **setNeutralValue**(const float neutralValue_)

Set the neutral potential value. The value should be in range 0.0 to 1.0, other values will be clamped to this range.

Deprecated:

Use *setNeutralPotential()* method instead.

See also:

dv::EdgeMapAccumulator::setNeutralPotential

Parameters
neutralValue_ – Neutral potential value in range [0.0; 1.0].

inline float **getNeutralPotential**() const

Get the neutral potential value for the accumulator. The range for potential value is [0.0; 1.0], where 1.0 stands for maximum possible potential - 255 in 8-bit pixel representation.

Returns
Neutral potential value in range [0.0; 1.0]

inline void **setNeutralPotential**(const float neutralPotential)

Set the neutral potential value. The value should be in range 0.0 to 1.0, other values will be clamped to this range.

Parameters
neutralPotential – Neutral potential value in range [0.0; 1.0].

inline float **getDecay**() const

Get current decay value.

Returns
Decay value.

inline void **setDecay**(const float decay_)

Set the decay value. Decay value is clamped to range of [0.0; 1.0].

Parameters
decay_ – Decay value. Negative value disabled the decay.

Public Static Attributes

static constexpr float **DECAY_NONE** = 0.0f

Decay coefficient value to disable any decay - zero decay.

static constexpr float **DECAY_FULL** = 1.0f

Maximum decay coefficient value which causes reset of pixels into neutral potential at each frame generation.

Protected Types

enum class **DecayMode**

Values:

enumerator **None**

enumerator **Full**

enumerator **Decay**

Protected Attributes

dv::*EventStore* **buffer**

Buffer to keep the latest events

uint8_t **maxByteValue** = 255

Max unsigned byte value

float **contribution** = 0.25f

Default contribution

uint8_t **drawIncrement** = (static_cast<uint8_t>(static_cast<float>(*maxByteValue*) * *contribution*))

Increment value for a single event

std::vector<uint8_t> **incrementLUT**

A look-up table for increment values at each possible pixel value.

bool **ignorePolarity** = true

float **neutralValue** = 0.f

uint8_t **neutralByteValue** = 0

float **decay** = 1.0

std::vector<uint8_t> **decayLUT**

cv::Mat **imageBuffer**

DecayMode **decayMode** = *DecayMode*::*Full*

struct **EigenEvents**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> A structure that contains events represented in eigen matrices. Useful for mathematical operations using the Eigen library..

Public Functions

inline explicit **EigenEvents**(const size_t size)

Public Members

Eigen::Matrix<int64_t, Eigen::Dynamic, 1> **timestamps**

Eigen::Matrix<int16_t, Eigen::Dynamic, 2> **coordinates**

Eigen::Matrix<uint8_t, Eigen::Dynamic, 1> **polarities**

struct **EmptyException**

Subclassed by *dv::exceptions::info::BadAlloc*, *dv::exceptions::info::IOError*, *dv::exceptions::info::LengthError*, *dv::exceptions::info::NullPointer*, *dv::exceptions::info::OutOfRange*, *dv::exceptions::info::RuntimeError*

Public Types

using **Info** = void

struct **EndOfFile**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **Epanechnikov**

Public Static Functions

static inline float **getSearchRadius**(const float bandwidth)

static inline float **apply**(const float squaredDistance, const float bandwidth)

struct **ErrorInfo**

Public Members

dv::*cstring* **mName**

dv::*cstring* **mTypeIdentifier**

struct **ErrorInfo**

Public Members

dv::*cstring* **mName**

dv::*cstring* **mTypeIdentifier**

class **EventBlobDetector**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/event_blob_detector.hpp> Event-based blob detector performing detection on accumulated event images.

Public Functions

inline explicit **EventBlobDetector**(const cv::Size &resolution, const int pyramidLevel = 0,
std::function<void(cv::Mat&)> preprocess = {},
cv::Ptr<cv::SimpleBlobDetector> blobDetector =
defaultBlobDetector())

Constructor for blob detector.

The detection steps are as following: 1) Compute accumulated image from events 2) Apply ROI to the accumulated event image 3) Down sample image (if pyramidLevel >= 1) 4) Apply preprocess function (if exists) 5) Detect blobs 6) Rescale blobs to original resolution (if pyramidLevel >= 1) 7) If ROI has an offset from (0,0) of initial image plane, add offset back to bring blobs location in the original image space coordinate system 8) Remove blobs where mask value is 0.

Parameters

- **resolution** – original image plane resolution
- **pyramidLevel** – integer defining number of down samples applied to the accumulated image. E.g. if pyramidLevel = 3 –> we down sample the image by a factor of 2 for N=3 times. this means that an image of size (100, 100) is down sampled to (25, 25) before performing the blob detection. Note that blob location is always returned in the original resolution size.
- **preprocess** – function to be applied to the accumulated image before performing the detection step. The function modifies the input image passed as argument to the function in place. Internally, the api check that resolution and type of the image are kept.
- **blobDetector** – blob detector instance performing the detection step

inline *dv*::*cvector*<*dv*::*TimedKeyPoint*> **detect**(const *dv*::*EventStore* &events, const cv::Rect &roi = cv::Rect(), const cv::Mat &mask = cv::Mat())

Detection step.

Parameters

- **events** – data used to create the accumulated image over which blob detection will be applied
- **roi** – region in which blobs will be searched
- **mask** – disable any blob detections on coordinates with zero pixel value on the mask.

Returns

blobs found from blob detector

Public Static Functions

static inline cv::Ptr<cv::SimpleBlobDetector> **defaultBlobDetector**()

Create a reasonable default blob detector.

The method creates an instance of cv::SimpleBlobDetector with following parameter values:

- filterByArea = true
- minArea = 10 : minimum area of blobs to be detected - reasonable value to safely detect blobs and not noise in the accumulated image
- maxArea = 10000
- filterByCircularity = false
- filterByConvexity = false
- filterByInertia = false

Returns

blob detector used by default to detect interesting blobs

Private Members

cv::Ptr<cv::SimpleBlobDetector> **mBlobDetector**

Blob detector instance performing the detection step

int32_t **mPyramidLevel**

Number of pyrDown applied to the accumulated image

std::function<void(cv::Mat&)> **mPreprocessFcn**

Preprocessing function to be applied before the detection step

dv::*EdgeMapAccumulator* **mAccumulator**

Accumulator generating the image used for blob detection

template<*dv*::*concepts*::*EventToFrameConverter*<*dv*::*EventStore*> **AccumulatorType** = *dv*::*EdgeMapAccumulator*>

class **EventCombinedLKTracker** : public *dv*::*features*::*ImageFeatureLKTracker*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/event_combined_lk_tracker.hpp> Implements an event combined Lucas-Kanade tracker. The algorithms detects and tracks features on a regular frame image, but to improve tracking quality, it accumulates intermediate frames from events, performs tracking on those frames and uses the output to predict the track locations on the regular frame.

Template Parameters
AccumulatorType – *Accumulator* class to be used for frame generation.

Public Types

using **SharedPtr** = *std*::shared_ptr<*EventCombinedLKTracker*>

using **UniquePtr** = *std*::unique_ptr<*EventCombinedLKTracker*>

Public Functions

inline void **accept**(const *dv*::*EventStore* &store)

Add an event batch. Added events should contain at least some events that were registered further in the future of the next image.

Parameters
store – Batch of events.

inline const *std*::vector<*std*::vector<cv::Point2f>> &**getEventTrackPoints**() const

Get the intermediate tracking points on the event frames.

Returns
A vector of tracked points on the intermediate frames.

inline const *std*::vector<*dv*::*features*::*ImagePyramid*> &**getAccumulatedFrames**() const

Get a vector containing the intermediate accumulated frames.

Returns
A vector containing the intermediate accumulated frames.

inline *dv*::*Duration* **getStoreTimeLimit**() const

Get the event storage time limit.

Returns
Duration of the event storage in microseconds.

inline void **setStoreTimeLimit**(const *dv*::*Duration* storeTimeLimit)

Set the event buffer storage duration limit.

Parameters
storeTimeLimit – Storage duration limit in microseconds.

inline size_t **getNumberOfEvents**() const

Get the number of latest events that are going to be accumulated for each frame.

Returns
Number of accumulated events.

inline void **setNumberOfEvents**(const size_t numberOfEvents)

Set the number of latest events that are going to be accumulated for each frame.

Parameters
_numberOfEvents – Number of accumulated events.

inline int **getNumIntermediateFrames**() const

Get the number of intermediate frames that are going to be generated.

Returns
Number of intermediate frames between the frames.

inline void **setNumIntermediateFrames**(const int numIntermediateFrames)

Set the number of intermediate frames that are going to be generated.

Parameters
numIntermediateFrames – Number of intermediate frames between the frames.

inline void **setAccumulator**(*std*::unique_ptr<*AccumulatorType*> accumulator)

Set an accumulator instance to be used for frame generation. If a `nullptr` is passed, the function will instantiate an accumulator with no parameters (defaults).

Parameters
accumulator – An accumulator instance, can be nullptr to instantiate a default accumulator.

inline virtual void **accept**(const *dv*::*measurements*::*Depth* &timedDepth) override

Add scene depth, a median depth value of tracked landmarks usually works well enough.

Parameters
timedDepth – Depth measurement value (pair of timestamp and measured depth)

inline virtual void **accept**(const *kinematics*::*Transformationf* &transform) override

Add camera transformation, usually in the world coordinate frame (`T_WC`). Although the class only extract the motion difference, so any other reference frame should also work as long as reference frames are not mixed up.

Parameters
transform – Camera pose represented by a transformation.

inline virtual void **accept**(const *dv*::*Frame* &image) override

Add an input image for the tracker. Image pyramid will be built from the given image.

Parameters
image – Acquired image.

inline double **getMinRateForIntermediateTracking**() const

Get the minimum event rate that is required to perform intermediate tracking.

Returns
Minimum event rate per second value.

inline void **setMinRateForIntermediateTracking**(const double minRateForIntermediateTracking)

Set a minimum event rate per second value that is used to perform intermediate. If the event rate between last and current frame is lower than this, tracker assumes very little motion and does not perform intermediate tracking.

Parameters
minRateForIntermediateTracking – Event rate (number of incoming events per second) required to perform intermediate tracking on accumulated frames.

inline virtual void **setConstantDepth**(const float depth) override

Set constant depth value that is assumed if no depth measurement is passed using `accept(dv::measurements::Depth)`. By default the constant depth is assumed to be 3.0 meters, which is just a reasonable guess.

This value is propagated into the accumulator if it supports constant depth setting.

Parameters
depth – Distance to the scene (depth).

Throws
`InvalidArgument` – Exception is thrown if a negative depth value is passed.

Public Static Functions

static inline *EventCombinedLKTracker*::*UniquePtr* **RegularTracker**(const cv::Size &resolution, const Config &config = Config(), *std*::unique_ptr<*AccumulatorType*> accumulator = nullptr, *ImagePyrFeatureDetector*::UniquePtr detector = nullptr, *RedetectionStrategy*::*UniquePtr* redetection = nullptr)

Create a tracker instance that performs tracking of features on both - event accumulated and regular images. Tracking is performed by detecting and tracking features on a regular image. It also uses events to generate intermediate accumulated frames between the regular frames, track the features on them and use the intermediate tracking results as feature position priors for the image frame.

Parameters

- **resolution** – Sensor resolution
- **config** – Lucas-Kanade tracker configuration
- **accumulator** – The accumulator instance to be used for intermediate frame accumulation. Uses *dv::EdgeMapAccumulator* with default parameters if `nullptr` is passed.
- **detector** – Feature (corner) detector to be used. Uses `cv::Fast` with a threshold of 10 by default.

- **redetection** – Feature redetection strategy. By default, redetects features when feature count is bellow 0.5 of maximum value.

Returns
The tracker instance

static inline *EventCombinedLKTracker*::*UniquePtr* **MotionAwareTracker**(const *camera*::CameraGeometry::SharedPtr &camera, const Config &config = Config(), *std*::unique_ptr<*AccumulatorType*> accumulator = nullptr, *kinematics*::*PixelMotionPredictor*::*UniquePtr* motionPredictor = nullptr, *ImagePyrFeatureDetector*::UniquePtr detector = nullptr, *RedetectionStrategy*::*UniquePtr* redetection = nullptr)

Create a tracker instance that performs tracking of features on both - event accumulated and regular images. Tracking is performed by detecting and tracking features on a regular image. It also uses events to generate intermediate accumulated frames between the regular frames, track the features on them and use the intermediate tracking results as feature position priors for the image frame. The implementation also uses camera motion and scene depth to motion compensate events, so the intermediate accumulated frames are sharp and the Lucas-Kanade tracker works more accurately. This requires camera sensor to be calibrated.

Parameters

- **camera** – Camera geometry class instance, containing the intrinsic calibration of the camera sensor.
- **config** – Lucas-Kanade tracker configuration
- **accumulator** – The accumulator instance to be used for intermediate frame accumulation. Uses *dv::EdgeMapAccumulator* with default parameters if `nullptr` is passed.
- **motionPredictor** – Motion predictor class, by default it uses pixel reprojection *dv::kinematics::PixelMotionPredictor* without distortion model.
- **detector** – Feature (corner) detector to be used. Uses `cv::Fast` with a threshold of 10 by default.
- **redetection** – Feature redetection strategy. By default, redetects features when feature count is bellow 0.5 of maximum value.

Returns
The tracker instance

Protected Functions

inline *std*::vector<cv::Point2f> **trackIntermediateEvents**()

Run the intermediate tracking on accumulated events. The lastFrameResults are modified if any of the intermediate tracks are lost. The predicted coordinates are returned which must match the indices of the keypoints in lastFrameResults keypoint list.

Returns
Predicted feature track locations that correspond to modified `lastFrameResults->keypoints` vector.

inline virtual Result::SharedPtr **track**() override

Perform the tracking.

Returns
Tracking result.

inline **EventCombinedLKTracker**(const cv::Size &resolution, const *ImageFeatureLKTracker*::*Config* &config)

Initialize the event combined Lucas-Kanade tracker and custom tracker parameters. It is going to use *EdgeMapAccumulator* with 15000 events and 0.25 event contribution. It will accumulate 3 intermediate frames from events to predict the track positions on regular frame.

Parameters

- **resolution** – Image resolution.
- **config** – Image tracker configuration.

inline **EventCombinedLKTracker**(const *camera*::CameraGeometry::SharedPtr &camera, const *ImageFeatureLKTracker*::*Config* &config)

Initialize the event combined Lucas-Kanade tracker and custom tracker parameters. It is going to use *EdgeMapAccumulator* with 15000 events and 0.25 event contribution. It will accumulate 3 intermediate frames from events to predict the track positions on regular frame.

Parameters

- **camera** – Camera geometry.
- **config** – Image tracker configuration.

Protected Attributes

std::unique_ptr<*AccumulatorType*> **mAccumulator** = nullptr

dv::*Duration* **mStoreTimeLimit** = *dv*::*Duration*(5000000)

size_t **mNumberOfEvents** = 20000

double **mMinRateForIntermediateTracking** = 0

int **mNumIntermediateFrames** = 3

dv::*EventStore* **mEventBuffer**

std::vector<*dv*::*features*::*ImagePyramid*> **mAccumulatedFrames**

std::vector<*std*::vector<cv::Point2f>> **mEventTrackPoints**

template<*concepts*::*EventToFrameConverter*<*dv*::*EventStore*> **AccumulatorType** = *dv*::*EdgeMapAccumulator*>

class **EventFeatureLKTracker** : public *dv*::*features*::*ImageFeatureLKTracker*

#include *</builds/inivation/dv/dv-processing/include/dv-processing/features/event_feature_lk_tracker.hpp>*
Event-based Lucas-Kanade tracker, the tracking is achieved by accumulating frames and running the classic LK frame based tracker on them.

Since the batch of events might contain information for more than a single tracking iteration configurable by the framerate parameter, the tracking function should be executed on a loop until it returns a null-pointer, signifying end of available data processing:

```
tracker.addEventInput(eventStore);
while (auto result = tracker.runTracking()) {
    // process the tracking result
}
```

Template Parameters
AccumulatorType – *Accumulator* class to be used for frame generation.

Public Types

using **SharedPtr** = *std*::shared_ptr<*EventFeatureLKTracker*>

using **UniquePtr** = *std*::unique_ptr<*EventFeatureLKTracker*>

Public Functions

inline const cv::Mat &**getAccumulatedFrame**() const

Get the latest accumulated frame.

Returns
An accumulated frame.

inline int **getFramerate**() const

Get configured framerate.

Returns
Current accumulation and tracking framerate.

inline void **setFramerate**(int framerate)

Set the accumulation and tracking framerate.

Parameters
framerate_ – New accumulation and tracking framerate.

inline void **accept**(const *dv*::*EventStore* &store)

Add the input events. Since the batch of events might contain information for more than a single tracking iteration configurable by the framerate parameter, the tracking function should be executed on a loop until it returns a null-pointer, signifying end of available data processing:

```
tracker.addEventInput(eventStore);
while (auto result = tracker.runTracking()) {
    // process the tracking result
}
```

Parameters
store – Event batch.

inline *dv*::*Duration* **getStoreTimeLimit**() const

Get the event storage time limit.

Returns
Duration of the event storage in microseconds.

inline void **setStoreTimeLimit**(const *dv*::*Duration* storeTimeLimit)

Set the event buffer storage duration limit.

Parameters
storeTimeLimit – Storage duration limit in microseconds.

inline size_t **getNumberOfEvents**() const

Get the number of latest events that are going to be accumulated for each frame. The default number of event is a third of of total pixels in the sensor.

Returns
Number of event to be accumulated.

inline void **setNumberOfEvents**(size_t numberOfEvents)

Set the number of latest events that are going to be accumulated for each frame. The default number of event is a third of of total pixels in the sensor.

Parameters
numberOfEvents – Number of accumulated events.

inline void **setAccumulator**(*std*::unique_ptr<*AccumulatorType*> accumulator)

Set an accumulator instance to be used for frame generation. If a `nullptr` is passed, the function will instantiate an accumulator with no parameters (defaults).

Parameters
accumulator – An accumulator instance, can be nullptr to instantiate a default accumulator.

inline virtual void **accept**(const *dv*::*measurements*::*Depth* &timedDepth) override

Add scene depth, a median depth value of tracked landmarks usually works well enough.

Parameters
timedDepth – Depth measurement value (pair of timestamp and measured depth)

inline virtual void **accept**(const *kinematics*::*Transformationf* &transform) override

Add camera transformation, usually in the world coordinate frame (`T_WC`). Although the class only extract the motion difference, so any other reference frame should also work as long as reference frames are not mixed up.

Parameters
transform – Camera pose represented by a transformation.

inline virtual void **setConstantDepth**(const float depth) override

Set constant depth value that is assumed if no depth measurement is passed using `accept(dv::measurements::Depth)`. By default the constant depth is assumed to be 3.0 meters, which is just a reasonable guess.

This value is used for predicting feature track positions when no depth measurements are passed in and also is propagated into the accumulator if it supports constant depth setting.

Parameters
depth – Distance to the scene (depth).

Throws

`InvalidArgument` – Exception is thrown if a negative depth value is passed.

Public Static Functions

static inline *EventFeatureLKTracker*::*UniquePtr* **RegularTracker**(const cv::Size &resolution, const Config &config = Config(), *std*::unique_ptr<*AccumulatorType*> accumulator = nullptr, *ImagePyrFeatureDetector*::UniquePtr detector = nullptr, *RedetectionStrategy*::*UniquePtr* redetection = nullptr)

Create a tracker instance that performs tracking of features on event accumulated frames. Features are detected and tracked on event accumulated frames.

Parameters

- **resolution** – Sensor resolution
- **config** – Lucas-Kanade tracker configuration
- **accumulator** – The accumulator instance to be used for intermediate frame accumulation. Uses *dv::EdgeMapAccumulator* with default parameters if `nullptr` is passed.
- **detector** – Feature (corner) detector to be used. Uses `cv::Fast` with a threshold of 10 by default.
- **redetection** – Feature redetection strategy. By default, redetects features when feature count is bellow 0.5 of maximum value.

Returns

The tracker instance

static inline *EventFeatureLKTracker*::*UniquePtr* **MotionAwareTracker**(const *camera*::CameraGeometry::SharedPtr &camera, const Config &config = Config(), *std*::unique_ptr<*AccumulatorType*> accumulator = nullptr, *kinematics*::*PixelMotionPredictor*::*UniquePtr* motionPredictor = nullptr, *ImagePyrFeatureDetector*::UniquePtr detector = nullptr, *RedetectionStrategy*::*UniquePtr* redetection = nullptr)

Create a tracker instance that performs tracking of features on event accumulated frames. Features are detected and tracked on event accumulated frames. Additionally, camera motion and scene depth are used to generate motion compensated frames, which are way sharper than usual accumulated frames. This requires camera sensor to be calibrated.

Parameters

- **camera** – Camera geometry class instance, containing the intrinsic calibration of the camera sensor.
- **config** – Lucas-Kanade tracker configuration

- **accumulator** – The accumulator instance to be used for intermediate frame accumulation. Uses *dv::EdgeMapAccumulator* with default parameters if `nullptr` is passed.
- **motionPredictor** – Motion predictor class, by default it uses pixel reprojection *dv::kinematics::PixelMotionPredictor* without distortion model.
- **detector** – Feature (corner) detector to be used. Uses `cv::Fast` with a threshold of 10 by default.
- **redetection** – Feature redetection strategy. By default, redetects features when feature count is bellow 0.5 of maximum value.

Returns
The tracker instance

Protected Functions

inline virtual Result::SharedPtr **track**() override

Perform the tracking

Returns
Tracking result.

inline explicit **EventFeatureLKTracker**(const cv::Size &dimensions, const Config &config)

Initialize the event-frame tracker with default configuration: all the defaults of *ImageFeatureLKTracker* and a *EdgeMapAccumulator* executing at 50 FPS with and event count equal to third of the camera resolution and event contribution of 0.25.

Parameters

- **imageDimensions** – Image resolution.
- **config** – Lukas-Kanade tracker configuration.

inline explicit **EventFeatureLKTracker**(const *dv*::*camera*::*CameraGeometry*::*SharedPtr* &camera, const Config &config)

Initialize the event-frame tracker with default configuration: all the defaults of *ImageFeatureLKTracker* and a *EdgeMapAccumulator* executing at 50 FPS with and event count equal to third of the camera resolution and event contribution of 0.25.

Parameters

- **camera** – Camera geometry.
- **config** – Lukas-Kanade tracker configuration.

Protected Attributes

std::unique_ptr<*AccumulatorType*> **mAccumulator** = nullptr

int **mFramerate** = 50

int64_t **mPeriod** = 1000000 / *mFramerate*

int64_t **mLastRunTimestamp** = 0

dv::*Duration* **mStoreTimeLimit** = *dv*::*Duration*(5000000)

size_t **mNumberOfEvents**

The default number of event is a third of of total pixels in the sensor.

dv::*EventStore* **mEventBuffer**

cv::Mat **mAccumulatedFrame**

Private Functions

inline virtual void **accept**(const *dv*::*Frame* &image)

Add an input image for the tracker. Image pyramid will be built from the given image.

Parameters
image – Acquired image.

inline virtual void **accept**(const *dv*::*measurements*::*Depth* &timedDepth)

Add scene depth, a median depth value of tracked landmarks usually works well enough.

Parameters
timedDepth – Depth measurement value (pair of timestamp and measured depth)

inline virtual void **accept**(const *dv*::*kinematics*::*Transformationf* &transform)

Add camera transformation, usually in the world coordinate frame (`T_WC`). Although the class only extract the motion difference, so any other reference frame should also work as long as reference frames are not mixed up.

Parameters
transform – Camera pose represented by a transformation

template<class **EventStoreClass** = *dv*::*EventStore*>

class **EventFilterBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/filters.hpp> A base class for noise filter implementations. Handles data input and output, derived classes only have to implement a retain function that tests whether event should be retained or discarded.

Subclassed by *dv::EventFilterChain< EventStoreClass >*, *dv::EventMaskFilter< EventStoreClass >*, *dv::EventPolarityFilter< EventStoreClass >*, *dv::EventRegionFilter< EventStoreClass >*, *dv::RefractoryPeriodFilter< EventStoreClass >*, *dv::noise::BackgroundActivityNoiseFilter< EventStoreClass >*, *dv::noise::FastDecayNoiseFilter< EventStoreClass >*

Public Functions

inline void **accept**(const *EventStoreClass* &store)

Accepts incoming events.

Parameters
store – Event packet.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept = 0

A function to be implemented by derived class which tests whether given event should be retained or discarded.

Parameters
event – An event to be checked.

Returns
Return true if the event is to be retained or false to discard the event.

inline *EventStoreClass* **generateEvents**()

Apply the filter algorithm and return only the filtered events from the ones that were accepted as input.

Returns

inline size_t **getNumIncomingEvents**() const

Get number of total events that were accepted by the noise filter.

Returns
Total number of incoming events to this filter instance.

inline size_t **getNumOutgoingEvents**() const

Total number of outgoing events from this filter instance.

Returns
Total number of outgoing events from this filter instance.

inline float **getReductionFactor**() const

Get the reduction factor of this filter. It's a fraction representation of events that were discard by this filter compared to the amount of incoming events.

Returns
Reduction factor value.

virtual **~EventFilterBase**() = default

inline *EventStoreClass* &**operator>>**(*EventStoreClass* &out)

Retrieve filtered events using output stream operator.

Parameters
out – Filtered events.

Returns

Protected Attributes

EventStoreClass **buffer**

int64_t **highestProcessedTime** = -1

size_t **numIncomingEvents** = 0

size_t **numOutgoingEvents** = 0

template<class **EventStoreClass** = *dv*::*EventStore*>

class **EventFilterChain** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/filters.hpp> Event filter based on multiple event filter applied sequentially. Internally stores any added filters and

Template Parameters
EventStoreClass – Type of event store

Public Functions

inline void **addFilter**(*std*::shared_ptr<*dv*::*EventFilterBase*<*EventStoreClass*>> filter)

Add a filter to the chain of filtering.

Parameters
filter –

inline *EventFilterChain* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters
events – Input events.

Returns

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept override

Test whether event is of configured polarity.

Parameters
event – Event to be checked.

Returns
True if event has the expected polarity, false otherwise.

Protected Attributes

std::vector<*std*::shared_ptr<*dv*::*EventFilterBase*<*EventStoreClass*>>> **filters**

template<class **EventStoreClass** = *dv*::*EventStore*>

class **EventMaskFilter** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

Public Functions

inline explicit **EventMaskFilter**(const cv::Mat &mask)

Create an event masking filter. Discards any events that happen on coordinates where mask has a zero value and retains all events with coordinates where mask has a non-zero value.

Parameters
mask – The mask to be applied (requires CV_8UC1 type).

Throws
`InvalidArgument` – Exception thrown if the mask is of incorrect type.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept override

A function to be implemented by derived class which tests whether given event should be retained or discarded.

Parameters

event – An event to be checked.

Returns

Return true if the event is to be retained or false to discard the event.

inline const cv::Mat &**getMask**() const

Get the mask that is currently applied.

Returns

inline void **setMask**(const cv::Mat &mask)

Set a new mask to this filter.

Parameters

mask – The mask to be applied (requires CV_8UC1 type).

inline *EventMaskFilter* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters

events – Input events.

Returns

Private Members

cv::Mat **mMask**

struct **EventPacket** : public *flatbuffers*::NativeTable

Public Types

typedef *EventPacketFlatbuffer* **TableType**

Public Functions

inline **EventPacket**()

inline **EventPacket**(const *dv*::*cvector*<Event> &_elements)

Public Members

dv::*cvector*<Event> **elements**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *EventPacket* &packet)

struct **EventPacketBuilder**

Public Functions

inline void **add_elements**(*flatbuffers*::Offset<*flatbuffers*::Vector<const Event*>> elements)

inline explicit **EventPacketBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

EventPacketBuilder &**operator=**(const *EventPacketBuilder*&)

inline *flatbuffers*::Offset<*EventPacketFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **EventPacketFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *EventPacket* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<const Event*> ***elements**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *EventPacket* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*EventPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*EventPacket* *_o, const *EventPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*EventPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *EventPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "EVTS"

template<class **EventStoreClass** = *dv*::*EventStore*>

class **EventPolarityFilter** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/filters.hpp> Event filter based on polarity.

Template Parameters
EventStoreClass – Type of event store

Public Functions

inline explicit **EventPolarityFilter**(const bool polarity)

Construct an event filter which filters out only events of given polarity.

Parameters
polarity – Extract events only of matching polarity.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept override

Test whether event is of configured polarity.

Parameters
event – Event to be checked.

Returns
True if event has the expected polarity, false otherwise.

inline *EventPolarityFilter* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters
events – Input events.

Returns

Protected Attributes

bool **polarity**

template<class **EventStoreClass** = *dv*::*EventStore*>

class **EventRegionFilter** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/filters.hpp> Event filter that filters events based on a given ROI.

Template Parameters
EventStoreClass – Type of event store

Public Functions

inline explicit **EventRegionFilter**(const cv::Rect &roi)

Filter event based on an ROI.

Parameters
roi – Region of interest, events outside of this region will be discarded.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept override

Test whether event belongs to an ROI.

Parameters
event – Event to be checked.

Returns
True if event belongs to ROI, false otherwise.

inline *EventRegionFilter* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters
events – Input events.

Returns

Protected Attributes

cv::Rect **roi**

template<*dv*::*concepts*::*AddressableEvent* **EventType**>

class **EventTimeComparator**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> **INTERNAL USE ONLY** Compares an events timestamp to that of a timestamp.

Public Functions

inline bool **operator()**(const *EventType* &evt, const int64_t time) const

inline bool **operator()**(const int64_t time, const *EventType* &evt) const

class **EventVisualizer**

#include </builds/inivation/dv/dv-processing/include/dv-processing/visualization/event_visualizer.hpp> *EventVisualizer* class implements simple color-coded representation of events. It applies certain colors where positive or negative polarity events are registered.

Public Functions

inline explicit **EventVisualizer**(const cv::Size &resolution, const cv::Scalar &backgroundColor = *colors*::*white*, const cv::Scalar &positiveColor = *colors*::*iniBlue*, const cv::Scalar &negativeColor = *colors*::*darkGrey*)

Initialize event visualizer.

Parameters

- **resolution** – Resolution of incoming events.
- **backgroundColor** – Background color.
- **positiveColor** – Color applied to positive polarity events.
- **negativeColor** – Color applied to negative polarity events.

inline cv::Mat **generateImage**(const *dv*::*EventStore* &events) const

Generate a preview image from an event store.

Parameters
events – Input events.

Returns
Colored preview image of given events.

inline void **generateImage**(const *dv*::*EventStore* &events, cv::Mat &background) const

Generate a preview image from an event store.

Parameters

- **events** – Input events.
- **background** – Image to draw the events on. The pixels type has to be 3-channel 8-bit unsigned integer (BGR).

inline cv::Scalar **getBackgroundColor**() const

Get currently configured background color.

Returns
Background color.

inline void **setBackgroundColor**(const cv::Scalar &backgroundColor_)

Set new background color.

Parameters
backgroundColor_ – New background color.

inline cv::Scalar **getPositiveColor**() const

Get currently configured positive polarity color.

Returns
Positive polarity color.

inline void **setPositiveColor**(const cv::Scalar &positiveColor_)

Set new positive polarity color.

Parameters
positiveColor_ – New positive polarity color.

inline cv::Scalar **getNegativeColor**() const

Get negative polarity color.

Returns
Negative polarity color.

inline void **setNegativeColor**(const cv::Scalar &negativeColor_)

Set new negative polarity color.

Parameters
negativeColor_ – New negative polarity color.

Private Members

const cv::Size **resolution**

cv::Vec3b **backgroundColor**

cv::Vec3b **positiveColor**

cv::Vec3b **negativeColor**

class **Exception** : public *std*::exception

Subclassed by *dv::exceptions::Exception_< EXCEPTION_TYPE, BASE_TYPE >*

Public Functions

inline explicit **Exception**(const *std*::source_location &location = *std*::source_location::current(), const boost::stacktrace::stacktrace &stacktrace = boost::stacktrace::stacktrace(), const *std*::string_view type = boost::core::demangle(typeid(*Exception*).name()))

inline explicit **Exception**(const *std*::string_view whatInfo, const *std*::source_location &location = *std*::source_location::current(), const boost::stacktrace::stacktrace &stacktrace = boost::stacktrace::stacktrace(), const *std*::string_view type = boost::core::demangle(typeid(*Exception*).name()))

~Exception() override = default

Exception(const *Exception* &other) = default

Exception(*Exception* &&other) = default

inline *Exception* **operator<<**(const *std*::string_view info)

inline const char ***what**() const noexcept override

Protected Attributes

std::string **mInfo**

Private Functions

inline void **createInfo**(const *std*::string_view whatInfo, const *std*::string_view file, const *std*::string_view function, const uint32_t line, const *std*::string_view stacktrace, const *std*::string_view type)

template<typename **EXCEPTION_TYPE**, typename **BASE_TYPE** = *Exception*>

class **Exception_** : public *dv*::*exceptions*::*Exception*

Public Types

using **Info** = typename *EXCEPTION_TYPE*::Info

Public Functions

template<*internal*::*HasExtraExceptionInfo* **T** = *EXCEPTION_TYPE*>
inline **Exception_**(const *std*::string_view whatInfo, const typename *T*::Info &errorInfo, const *std*::source_location &location = *std*::source_location::current(), const boost::stacktrace::stacktrace &stacktrace = boost::stacktrace::stacktrace(), const *std*::string_view type = boost::core::demangle(typeid(*EXCEPTION_TYPE*).name()))

template<*internal*::*HasExtraExceptionInfo* **T** = *EXCEPTION_TYPE*>

inline **Exception_**(const typename *T*::Info &errorInfo, const *std*::source_location &location = *std*::source_location::current(), const boost::stacktrace::stacktrace &stacktrace = boost::stacktrace::stacktrace(), const *std*::string_view type = boost::core::demangle(typeid(*EXCEPTION_TYPE*).name()))

inline **Exception_**(const *std*::string_view whatInfo, const *std*::source_location &location = *std*::source_location::current(), const boost::stacktrace::stacktrace &stacktrace = boost::stacktrace::stacktrace(), const *std*::string_view type = boost::core::demangle(typeid(*EXCEPTION_TYPE*).name()))

inline **Exception_**(const *std*::source_location &location = *std*::source_location::current(), const boost::stacktrace::stacktrace &stacktrace = boost::stacktrace::stacktrace(), const *std*::string_view type = boost::core::demangle(typeid(*EXCEPTION_TYPE*).name()))

~Exception_() override = default

Exception_(const *Exception_* &other) = default

Exception_(*Exception_* &&other) = default

template<*internal*::*HasExtraExceptionInfo* **T** = *EXCEPTION_TYPE*>
inline *Exception_* **operator<<**(const typename *T*::Info &errorInfo)

inline *Exception_* **operator<<**(const *std*::string_view whatInfo)

template<class **EventStoreClass** = *dv*::*EventStore*>

class **FastDecayNoiseFilter** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

Public Functions

inline explicit **FastDecayNoiseFilter**(const cv::Size &resolution, const *dv*::*Duration* halfLife = `dv::Duration(10 '000)`, const int subdivisionFactor = 4, const float noiseThreshold = 6.f)

Create a fast decay noise filter. This filter uses a concept that performs a fast decay on a low resolution representation of the image and checks whether corresponding neighbourhood of the event has recent activity.

Parameters

- **resolution** – Sensor resolution.
- **halfLife** – Half-life is the amount of time it takes for the internal event counter to halve. Decreasing this will increase the strength of the noise filter (cause it to reject more events).
- **subdivisionFactor** – Subdivision factor, this is used calculate a low resolution image dimensions used for the fast decay operations.
- **noiseThreshold** – Noise threshold value, amount of filtered events can be increased by decreasing this value.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept override

Test whether to retain this event.

Parameters
event – Event to be checked.

Returns
True to retain an event, false to discard it.

inline *FastDecayNoiseFilter* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters
events – Input events.

Returns

inline float **getNoiseThreshold**() const

Get the currently configured noise threshold.

Returns
Noise threshold value.

inline void **setNoiseThreshold**(const float noiseThreshold)

Set a new noise threshold value.

Parameters
noiseThreshold – Noise threshold value.

inline *dv*::*Duration* **getHalfLife**() const

Get the current configured half-life value.

Half-life is the amount of time it takes for the internal event counter to halve. Decreasing this will increase the strength of the noise filter (cause it to reject more events).

Returns
Currently configured event counter half life value.

inline void **setHalfLife**(const *dv*::*Duration* halfLife)

Set a new counter half-life value.

Half-life is the amount of time it takes for the internal event counter to halve. Decreasing this will increase the strength of the noise filter (cause it to reject more events).

Parameters
halfLife – New event counter half life value.

Private Members

int **mSubdivisionFactor** = 4

cv::Mat **mDecayLUT**

dv::*TimeSurface* **mTimeSurface**

float **mNoiseThreshold** = 6.f

float **mHalfLifeMicros** = 10'000.f

class **FeatureCountRedetection** : public *dv*::*features*::*RedetectionStrategy*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/redetection_strategy.hpp> Redetection strategy based on number of features.

Public Functions

inline explicit **FeatureCountRedetection**(float minimumProportionOfTracks)

Redetection strategy based on number of features.

Parameters

minimumProportionOfTracks – Feature count coefficient, redetection is performed when feature count goes lower than the given proportion of maximum tracks, redetection will be executed.

inline virtual bool **decideRedetection**(const *TrackerBase* &tracker) override

Check whether to perform redetection.

Parameters

tracker – Current state of the tracker.

Returns

True to perform redetection of features, false to continue.

Protected Attributes

float **mMinimumProportionOfTracks** = 0.5f

template<class **InputType**, *dv*::*concepts*::*FeatureDetectorAlgorithm*<*InputType*> **Algorithm**>

class **FeatureDetector**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/feature_detector.hpp> A base class to implement feature detectors on different input types, specifically either images, time surfaces, or event stores. The implementing class should override the `detect` function and output a vector of unordered features with a quality score. The API will handle margin calculations and post processing of the features.

See also:

dv::concepts::FeatureDetectorAlgorithm

Template Parameters

- **InputType** – The type of input that is needed for the detector.
- **Algorithm** – The underlying detection algorithm, can be an OpenCV::Feature2D algorithm or a custom implementation, as long as it satisfies

Public Types

enum class **FeaturePostProcessing**

Feature post processing step performed after the the features were detected. Currently available types of post processing:

- None: Do not perform any postprocessing, all keypoints from detection will be returned.
- TopN: Retrieve the top number of highest scoring features.
- AdaptiveNMS: Apply the AdaptiveNMS algorithm to retrieve equally spaced keypoints in pixel space dimensions. More information on the AdaptiveNMS here: original code: https://github.com/BAILOOL/ANMS-Codes paper: https://www.researchgate.net/publication/323388062_Efficient_adaptive_non-maximal_suppression_algorithms_for_homogeneous_spatial_keypoint_distribution

Values:

enumerator **None**

enumerator **TopN**

enumerator **AdaptiveNMS**

using **ThisType** = *FeatureDetector*<*InputType*, *Algorithm*>

using **SharedPtr** = *std*::shared_ptr<*ThisType*>

using **UniquePtr** = *std*::unique_ptr<*ThisType*>

using **AlgorithmPtr** = typename *std*::conditional_t<*std*::is_base_of_v<cv::Feature2D, *Algorithm*>, cv::Ptr<*Algorithm*>, *std*::shared_ptr<*Algorithm*>>

Public Functions

inline **FeatureDetector**(const cv::Size &_imageDimensions, const *AlgorithmPtr* &_detector, *FeaturePostProcessing* _postProcessing, float _margin = 0.02f)

Create a feature detector.

See also:

FeatureDetectorBase::FeaturePostProcessing

Parameters

- **_imageDimensions** – Image dimensions.
- **_postProcessing** – Post processing step - subsampling of events,
- **_margin** – Margin coefficient, it will be multiplied by the width and height of the image to calculate an adaptive border alongside the edges of image, where features should not be detected.

inline explicit **FeatureDetector**(const cv::Size &_imageDimensions, const *AlgorithmPtr* &_detector)

Create a feature detector. This constructor defaults post-processing step to AdaptiveNMS and margin coefficient value of 0.02.

Parameters

_imageDimensions – Image dimensions.

virtual **~FeatureDetector**() = default

Destructor

inline *dv*::*cvector*<*dv*::*TimedKeyPoint*> **runDetection**(const *InputType* &input, size_t numPoints, const cv::Mat &mask = cv::Mat())

Public detection call. Calls the overloaded `detect` function, applies margin and post processing.

Parameters

- **input** – The input to the detector

- **numPoints** – Number of keypoints to be detected
- **mask** – Detection mask, detection will be performed where mask value is non-zero.

Returns
A list of keypoints with timestamp.

inline void **runRedetection**(*dv*::*cvector*<*dv*::*TimedKeyPoint*> &prior, const *InputType* &input, size_t numPoints, const cv::Mat &mask = cv::Mat())

Redetect new features and add them to already detected features. This function performs detection within masked region (if mask is non-empty), runs postprocessing and appends the additional features to the prior keypoint list.

Parameters

- **prior** – A list of existing features.
- **input** – The input to the detector (events, images, etc.).
- **numPoints** – Number of total features after detection.
- **mask** – Detection mask.

inline *FeaturePostProcessing* **getPostProcessing**() const

Get the type of post-processing.

See also:

FeatureDetectorBase::FeaturePostProcessing

Returns
Type of post-processing.

inline void **setPostProcessing**(*FeaturePostProcessing* _postProcessing)

Set the type of post-processing.

See also:

FeatureDetectorBase::FeaturePostProcessing

Parameters
_postProcessing – Type of post-processing.

inline float **getMargin**() const

Get currently applied margin coefficient. Margin coefficient is multiplied by the width and height of the image to calculate an adaptive border alongside the edges of image, where features should not be detected.

Returns
The margin coefficient.

inline void **setMargin**(float _margin)

Set the margin coefficient. Margin coefficient is multiplied by the width and height of the image to calculate an adaptive border alongside the edges of image, where features should not be detected.

Parameters
_margin – The margin coefficient

inline bool **isWithinROI**(const cv::Point2f &point) const

Check whether a point belongs to the ROI without the margins.

Parameters
point – Point to be checked

Returns
True if point belongs to the valid ROI, false otherwise.

inline const cv::Size &**getImageDimensions**() const

Get configured image dimensions.

Returns
Image dimensions.

Private Functions

inline *dv*::*cvector*<*dv*::*TimedKeyPoint*> **detect**(const *InputType* &input, const cv::Rect &roi, const cv::Mat &mask)

The detection function to be implemented for feature detection. It should return a list of keypoints with a quality score, but it should *not* be ordered in any way. The sorting will be performed by the `runDetection` function as a postprocessing step.

Parameters

- **input** – Input for the detector.
- **roi** – Region of interest where detection should be performed, the region is estimated using the margin configuration value.
- **mask** – Detection mask, can be empty. If non empty, the detection should be performed where mask value is non-zero.

Returns
A list of keypoint features with timestamp.

inline cv::Rect **getMarginROI**() const

Calculate the region of interest with the margin coefficient. Margin is a coefficient of width / height, which should be used to ignore pixels near borders of the image.

Returns
Region of interest for detection of features.

Private Members

FeaturePostProcessing **postProcessing**

float **margin**

cv::Size **imageDimensions**

cv::Rect **roiBuffered**

AlgorithmPtr **detector**

Container of the feature detector

int **classIdCounter** = 0

Class id counter, each new feature will be assigned on incremented class id.

KeyPointResampler **resampler**

class **FeatureTracks**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/feature_tracks.hpp> A class to store a time limited amount of feature tracks. Sorts and stores the data in separate queues for each track id. Provides `visualize` function to generate visualization images of the tracks.

Public Functions

inline void **accept**(const *dv*::*TimedKeyPoint* &keypoint)

Add a keypoint measurement into the feature track.

Parameters
keypoint – Single keypoint measurement.

inline void **accept**(const *dv*::*TimedKeyPointPacket* &keypoints)

Add a set of keypoint measurements into the feature track.

Parameters
keypoints – Vector of keypoint measurements.

inline void **accept**(const cv::KeyPoint &keypoint)

Add OpenCV type keypoint. It is missing a timestamp, so current system clock time will be used for the timestamp.

Parameters
keypoint – KeyPoint measurement.

inline void **accept**(const *TrackerBase*::*Result*::*ConstPtr* &trackingResult)

Add keypoint tracking result from a tracker.

Parameters
trackingResult – Tracking results.

inline *Duration* **getHistoryDuration**() const

Retrieve the history duration.

Returns
Currently applied track history time limit.

inline void **setHistoryDuration**(const *dv*::*Duration* historyDuration)

Set new history duration limit to buffer. If the new limit is shorter than the previously set, the tracks will be reduced to the new limit right away.

Parameters
historyDuration – New time limit for the track history buffer.

inline *std*::optional<*std*::shared_ptr<const *std*::deque<*dv*::*TimedKeyPoint*>>> **getTrack**(const int32_t trackId) const

Retrieve a track of given track id.

Parameters
trackId – Track id to retrieve.

Returns
A pointer to feature track history, `std::nullopt` if unavailable.

inline *std*::vector<int32_t> **getTrackIds**() const

Return all track ids that are available in the buffer.

Returns
A vector containing track ids store in the history buffer.

inline *dv*::*TimedKeyPointPacket* **getLatestTrackKeypoints**()

Return last keypoint from all tracks in the history.

Returns

inline void **eachTrack**(const *std*::function<void(const int32_t, const *std*::shared_ptr<const *std*::deque<*dv*::*TimedKeyPoint*>>&)> &callback) const

Run a callback function to each of the stored tracks.

Parameters
callback – Callback function that is going to be called for each of the tracks, tracks are passed into the callback function as arguments.

inline cv::Mat **visualize**(const cv::Mat &background) const

Draws tracks on the input image, by default uses neon color palette from the `dv::visualization::colors` namespace for each of the tracks.

Parameters
background – Background image to be used for tracks.

Throws
`InvalidArgument` – An `InvalidArgument` exception is thrown if an empty image is passed as background.

Returns
Input image with drawn colored feature tracks.

inline bool **isEmpty**() const

Checks whether the feature track history buffer is empty.

Returns
True if there are no feature keypoints in the buffer.

inline void **clear**()

Deletes any data stored in feature track buffer and resets visualization image.

inline const *std*::optional<*dv*::*Duration*> &**getTrackTimeout**() const

Get the track timeout value.

See also:

setTrackTimeout

Returns
Current track timeout value.

inline void **setTrackTimeout**(const *std*::optional<*dv*::*Duration*> &trackTimeout)

Set the track timeout value, pass `std::nullopt` to disable the this feature at all. Track latest timestamp is going to be compared to highest received timestamp in accept method, if the value is exceeded the track is going to be removed. This is useful to remove lost tracks without waiting for the track history to remove it, consider setting it to 2x of tracking rate, so tracks will remove if the track is not updated for two consecutive frames.

By default the feature is disabled, so lost tracks are kept until it's removed by the history time limit.

Parameters
trackTimeout – Track timeout value or `std::nullopt` to disable the feature.

inline int64_t **getHighestTime**()

Return latest time from all existing tracks.

Private Functions

inline void **addKeypoint**(const *dv*::*TimedKeyPoint* &keypoint)

Add a keypoint measurement

Parameters
keypoint – Keypoint measurement

inline void **maintainBufferDuration**()

Check the whole buffer for out-of-limit data, remove any tracks that do not contain any measurements.

Private Members

std::map<int32_t, *std*::shared_ptr<*std*::deque<*dv*::*TimedKeyPoint*>>> **mHistory**

dv::*Duration* **mHistoryDuration** = *dv*::*Duration*(500'000)

std::optional<*dv*::*Duration*> **mTrackTimeout** = *std*::nullopt

int64_t **mHighestTime** = -1

struct **FileDataDefinition** : public *flatbuffers*::NativeTable

Public Types

typedef *FileDataDefinitionFlatbuffer* **TableType**

Public Functions

inline **FileDataDefinition**()

inline **FileDataDefinition**(int64_t _ByteOffset, const PacketHeader &_PacketInfo, int64_t _NumElements, int64_t _TimestampStart, int64_t _TimestampEnd)

Public Members

int64_t **ByteOffset**

PacketHeader **PacketInfo**

int64_t **NumElements**

int64_t **TimestampStart**

int64_t **TimestampEnd**

Public Static Functions

static inline constexpr const char *__GetFullyQualifiedName__()

struct **FileDataDefinitionBuilder**

Public Functions

inline void **add_ByteOffset**(int64_t ByteOffset)

inline void **add_PacketInfo**(const PacketHeader *PacketInfo)

inline void **add_NumElements**(int64_t NumElements)

inline void **add_TimestampStart**(int64_t TimestampStart)

inline void **add_TimestampEnd**(int64_t TimestampEnd)

inline explicit **FileDataDefinitionBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

FileDataDefinitionBuilder &**operator=**(const *FileDataDefinitionBuilder*&)

inline *flatbuffers*::Offset<*FileDataDefinitionFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **FileDataDefinitionFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *FileDataDefinition* **NativeTableType**

Public Functions

inline int64_t **ByteOffset**() const

inline const PacketHeader ***PacketInfo**() const

inline int64_t **NumElements**() const

inline int64_t **TimestampStart**() const

inline int64_t **TimestampEnd**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *FileDataDefinition* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*FileDataDefinition* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*FileDataDefinition* *_o, const *FileDataDefinitionFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*FileDataDefinitionFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *FileDataDefinition* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

struct **FileDataTable** : public *flatbuffers*::NativeTable

Public Types

typedef *FileDataTableFlatbuffer* **TableType**

Public Functions

inline **FileDataTable**()

inline **FileDataTable**(const *dv*::*cvector*<*FileDataDefinition*> &_Table)

Public Members

dv::*cvector*<*FileDataDefinition*> **Table**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

struct **FileDataTableBuilder**

Public Functions

inline void **add_Table**(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*FileDataDefinitionFlatbuffer*>>> Table)

inline explicit **FileDataTableBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

FileDataTableBuilder &**operator=**(const *FileDataTableBuilder*&)

inline *flatbuffers*::Offset<*FileDataTableFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **FileDataTableFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *FileDataTable* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*FileDataDefinitionFlatbuffer*>> ***Table**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *FileDataTable* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*FileDataTable* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*FileDataTable* *_o, const *FileDataTableFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*FileDataTableFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *FileDataTable* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "FTAB"

struct **FileError**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **FileInfo**

Public Members

uint64_t **mFileSize**

dv::*CompressionType* **mCompression**

int64_t **mDataTablePosition**

int64_t **mDataTableSize**

dv::*FileDataTable* **mDataTable**

int64_t **mTimeLowest**

int64_t **mTimeHighest**

int64_t **mTimeDifference**

int64_t **mTimeShift**

std::vector<*dv*::*io*::*Stream*> **mStreams**

std::unordered_map<int32_t, *dv*::*FileDataTable*> **mPerStreamDataTables**

struct **FileNotFound**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **FileOpenError**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **FileReadError**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

struct **FileWriteError**

Public Types

using **Info** = *std*::filesystem::path

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

template<typename **T**>

struct **formatter**<*dv*::*basic_cstring*<*T*>> : public *fmt*::formatter<*std*::basic_string_view<*T*>>

Public Functions

template<typename **FormatContext**>
inline auto **format**(const *dv*::*basic_cstring*<*T*> &str, *FormatContext* &ctx)

template<>

struct **formatter**<*dv*::*BoundingBoxPacket*> : public *fmt*::ostream_formatter

template<typename **T**>

class **formatter**<*dv*::*cvector*<*T*>>

Public Functions

inline constexpr auto **parse**(format_parse_context &ctx)

template<typename **FormatContext**>
inline auto **format**(const *dv*::*cvector*<*T*> &vec, *FormatContext* &ctx)

Private Members

std::array<char, *FORMATTER_MAX_LEN*> **mFmtForward**

std::array<char, *FORMATTER_MAX_LEN*> **mSeparator**

Private Static Attributes

static constexpr size_t **FORMATTER_MAX_LEN** = {32}

template<>

struct **formatter**<*dv*::*DepthEventPacket*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*DepthFrame*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*EventPacket*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*EventStore*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*Frame*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*IMUPacket*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*io*::*CameraCapture*::*DVXeFPS*> : public *fmt*::ostream_formatter

template<>

class **formatter**<*dv*::*io*::*support*::*VariantValueOwning*>

Public Functions

inline constexpr auto **parse**(const format_parse_context &ctx)

template<typename **FormatContext**>
inline auto **format**(const *dv*::*io*::*support*::*VariantValueOwning* &obj, *FormatContext* &ctx)

Private Members

std::array<char, *FORMATTER_MAX_LEN*> **mFmtForward**

Private Static Attributes

static constexpr size_t **FORMATTER_MAX_LEN** = {32}

template<>

struct **formatter**<*dv*::*LandmarksPacket*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*Pose*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*TimedKeyPointPacket*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*dv*::*TriggerPacket*> : public *fmt*::ostream_formatter

template<>

struct **formatter**<*std*::filesystem::path> : public *fmt*::formatter<*std*::string>

Public Functions

template<typename **FormatContext**>
inline auto **format**(const *std*::filesystem::path &path, *FormatContext* &ctx)

template<typename **T**>

class **formatter**<*std*::vector<*T*>>

Public Functions

inline constexpr auto **parse**(format_parse_context &ctx)

template<typename **FormatContext**>
inline auto **format**(const *std*::vector<*T*> &vec, *FormatContext* &ctx)

Private Members

std::array<char, *FORMATTER_MAX_LEN*> **mFmtForward**

std::array<char, *FORMATTER_MAX_LEN*> **mSeparator**

Private Static Attributes

static constexpr size_t **FORMATTER_MAX_LEN** = {32}

struct **Frame** : public *flatbuffers*::NativeTable

Public Types

typedef *FrameFlatbuffer* **TableType**

Public Functions

inline **Frame**()

inline **Frame**(int64_t _timestamp, int64_t _timestampStartOfFrame, int64_t _timestampEndOfFrame, int64_t _timestampStartOfExposure, int64_t _timestampEndOfExposure, *FrameFormat* _format, int16_t _sizeX, int16_t _sizeY, int16_t _positionX, int16_t _positionY, const *dv*::*cvector*<uint8_t> &_pixels)

inline **Frame**(int64_t _timestamp, int64_t _exposure, int16_t _positionX, int16_t _positionY, const cv::Mat &_image, *dv*::*FrameSource* _source)

inline **Frame**(int64_t _timestamp, const cv::Mat &_image)

Public Members

int64_t **timestamp**

int16_t **positionX**

int16_t **positionY**

cv::Mat **image**

dv::*Duration* **exposure**

FrameSource **source**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *Frame* &frame)

struct **FrameBuilder**

Public Functions

inline void **add_timestamp**(int64_t timestamp)

inline void **add_timestampStartOfFrame**(int64_t timestampStartOfFrame)

inline void **add_timestampEndOfFrame**(int64_t timestampEndOfFrame)

inline void **add_timestampStartOfExposure**(int64_t timestampStartOfExposure)

inline void **add_timestampEndOfExposure**(int64_t timestampEndOfExposure)

inline void **add_format**(*FrameFormat* format)

inline void **add_sizeX**(int16_t sizeX)

inline void **add_sizeY**(int16_t sizeY)

inline void **add_positionX**(int16_t positionX)

inline void **add_positionY**(int16_t positionY)

inline void **add_pixels**(*flatbuffers*::Offset<*flatbuffers*::Vector<uint8_t>> pixels)

inline void **add_exposure**(int64_t exposure)

inline void **add_source**(*FrameSource* source)

inline explicit **FrameBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

FrameBuilder &**operator=**(const *FrameBuilder*&)

inline *flatbuffers*::Offset<*FrameFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **FrameFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *Frame* **NativeTableType**

Public Functions

inline int64_t **timestamp**() const
Central timestamp (µs), corresponds to exposure midpoint.

inline int64_t **timestampStartOfFrame**() const
Start of *Frame* (SOF) timestamp.

inline int64_t **timestampEndOfFrame**() const
End of *Frame* (EOF) timestamp.

inline int64_t **timestampStartOfExposure**() const
Start of Exposure (SOE) timestamp.

inline int64_t **timestampEndOfExposure**() const
End of Exposure (EOE) timestamp.

inline *FrameFormat* **format**() const
Pixel format (grayscale, RGB, …).

inline int16_t **sizeX**() const
X axis length in pixels.

inline int16_t **sizeY**() const
Y axis length in pixels.

inline int16_t **positionX**() const
X axis position (upper left offset) in pixels.

inline int16_t **positionY**() const
Y axis position (upper left offset) in pixels.

inline const *flatbuffers*::Vector<uint8_t> ***pixels**() const
Pixel values, 8bit depth.

inline int64_t **exposure**() const
Exposure duration.

inline *FrameSource* **source**() const
Source of the image data, whether it's from sensor or from some form of event accumulation.

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *Frame* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*Frame* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*Frame* *_o, const *FrameFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*FrameFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Frame* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "FRME"

struct **Gaussian**

Public Static Functions

static inline float **getSearchRadius**(const float bandwidth)

static inline float **apply**(const float squaredDistance, const float bandwidth)

class **ImageFeatureLKTracker** : public *dv*::*features*::*TrackerBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/image_feature_lk_tracker.hpp> A feature based sparse Lucas-Kanade feature tracker based on image pyramids.

Subclassed by *dv::features::EventCombinedLKTracker< AccumulatorType >*, *dv::features::EventFeatureLKTracker< AccumulatorType >*

Public Types

using **Config** = *LucasKanadeConfig*

using **SharedPtr** = *std*::shared_ptr<*ImageFeatureLKTracker*>

using **UniquePtr** = *std*::unique_ptr<*ImageFeatureLKTracker*>

Public Functions

inline virtual void **accept**(const *dv*::*Frame* &image)

Add an input image for the tracker. Image pyramid will be built from the given image.

Parameters
image – Acquired image.

inline void **setRedectionStrategy**(*RedetectionStrategy*::*UniquePtr* redetectionStrategy)

Set a new redetection strategy.

Deprecated:

Use setRedetectionStrategy instead

Parameters
redetectionStrategy – Redetection strategy instance.

inline void **setRedetectionStrategy**(*RedetectionStrategy*::*UniquePtr* redetectionStrategy)

Set a new redetection strategy.

Parameters
redetectionStrategy – Redetection strategy instance.

inline void **setDetector**(*ImagePyrFeatureDetector*::UniquePtr detector)

Set a new feature (corner) detector. If a `nullptr` is passed, the function will instantiate a feature detector with no parameters (defaults).

Parameters
detector – Feature detector instance.

inline void **setMotionPredictor**(*kinematics*::*PixelMotionPredictor*::*UniquePtr* predictor)

Set new pixel motion predictor instance. If a `nullptr` is passed, the function will instantiate a pixel motion predictor with no parameters (defaults).

Warning: motion prediction requires camera calibration to be set, otherwise the function will not instantiate the motion predictor.

Parameters
predictor – Pixel motion predictor instance.

inline virtual void **accept**(const *dv*::*measurements*::*Depth* &timedDepth)

Add scene depth, a median depth value of tracked landmarks usually works well enough.

Parameters
timedDepth – Depth measurement value (pair of timestamp and measured depth)

inline virtual void **accept**(const *dv*::*kinematics*::*Transformationf* &transform)

Add camera transformation, usually in the world coordinate frame (`T_WC`). Although the class only extract the motion difference, so any other reference frame should also work as long as reference frames are not mixed up.

Parameters
transform – Camera pose represented by a transformation

inline bool **isLookbackRejectionEnabled**() const

Check whether lookback is enabled.

Returns
True if lookback rejection is enabled.

inline void **setLookbackRejection**(const bool lookbackRejection)

Enable or disable lookback rejection based on Forward-Backward error. Lookback rejection applies Lucas-Kanade tracking backwards after running the usual tracking and rejects any tracks that fails to successfully track back to same approximate location by measuring Euclidean distance. Euclidean distance threshold for rejection can be set using `setRejectionDistanceThreshold` method.

This is a real-time implementation of the method proposed by Zdenek et al. 2010, that only performs forward-backward error measurement within a single pair of latest and previous frame: http://kahlan.eps.surrey.ac.uk/featurespace/tld/Publications/2010_icpr.pdf

See also:

setRejectionDistanceThreshold

Parameters

lookbackRejection – Pass true to enable lookback rejection based on Forward-Backward error.

inline float **getRejectionDistanceThreshold**() const

Get the current rejection distance threshold for the lookback rejection feature.

See also:

setRejectionDistanceThreshold

Returns

Rejection distance value which represents the Euclidean distance in pixel space between backward tracked feature pose and initial feature position before performing forward tracking.

inline void **setRejectionDistanceThreshold**(const float rejectionDistanceThreshold)

Set the threshold for lookback rejection feature. This value is a maximum Euclidean distance value that is considered successful when performing backwards tracking check after forward tracking. If the backward tracked feature location is further away from initial position than this given value, the tracker will reject the track as a failed track. See method `setLookbackRejection` documentation for further explanation of the approach.

See also:

setLookbackRejection

Parameters

rejectionDistanceThreshold – Rejection distance threshold value.

inline float **getConstantDepth**() const

Get currently assumed constant depth value. It is used if no depth measurements are provided.

See also:

setConstantDepth

Returns

Currently used aistance to the scene (depth).

inline virtual void **setConstantDepth**(const float depth)

Set constant depth value that is assumed if no depth measurement is passed using `accept(dv::measurements::Depth)`. By default the constant depth is assumed to be 3.0 meters, which is just a reasonable guess.

This value is used for predicting feature track positions when no depth measurements are passed in.

Parameters
depth – Distance to the scene (depth).

Throws
InvalidArgument – Exception is thrown if a negative depth value is passed.

Public Static Functions

static inline *ImageFeatureLKTracker*::*UniquePtr* **RegularTracker**(const cv::Size &resolution, const *Config* &_config = *Config*(), *ImagePyrFeatureDetector*::UniquePtr detector = nullptr, *RedetectionStrategy*::*UniquePtr* redetection = nullptr)

static inline *ImageFeatureLKTracker*::*UniquePtr* **MotionAwareTracker**(const *camera*::CameraGeometry::SharedPtr &camera, const *Config* &config = *Config*(), *kinematics*::*PixelMotionPredictor*::*UniquePtr* motionPredictor = nullptr, *ImagePyrFeatureDetector*::UniquePtr detector = nullptr, *RedetectionStrategy*::*UniquePtr* redetection = nullptr)

Protected Functions

inline *std*::vector<cv::Point2f> **predictNextPoints**(const int64_t previousTime, const *std*::vector<cv::Point2f> &previousPoints, const int64_t nextTime)

inline virtual Result::SharedPtr **track**() override

Perform the LK tracking.

Returns
Result of the tracking.

inline **ImageFeatureLKTracker**(const cv::Size &resolution, const *Config* &config)

Construct a tracker with default detector parameters, but configurable tracker parameters.

Parameters

- **resolution** – Image resolution.
- **_config** – Lucas-Kanade tracker parameters.

inline **ImageFeatureLKTracker**(const *camera*::*CameraGeometry*::*SharedPtr* &cameraGeometry, const *Config* &config)

Construct a tracker with default detector parameters, but configurable tracker parameters.

Parameters

- **resolution** – Image resolution.

- **_config** – Lucas-Kanade tracker parameters.

Protected Attributes

Config **mConfig** = {}

RedetectionStrategy::*UniquePtr* **mRedetectionStrategy** = nullptr

ImagePyrFeatureDetector::UniquePtr **mDetector** = nullptr

cv::Ptr<cv::SparsePyrLKOpticalFlow> **mTracker**

ImagePyramid::*UniquePtr* **mPreviousFrame** = nullptr

ImagePyramid::*UniquePtr* **mCurrentFrame** = nullptr

kinematics::*PixelMotionPredictor*::*UniquePtr* **mPredictor** = nullptr

std::unique_ptr<*kinematics*::*LinearTransformerf*> **mTransformer** = nullptr

std::map<int64_t, float> **mDepthHistory**

camera::*CameraGeometry*::*SharedPtr* **mCamera** = nullptr

cv::Size **mResolution**

bool **mLookbackRejection** = false

float **mRejectionDistanceThreshold** = 10.f

const int64_t **depthHistoryDuration** = 5000000

float **constantDepth** = 3.f

class **ImagePyramid**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/image_pyramid.hpp> Class that holds image pyramid layers with an according timestamp.

Public Types

typedef *std*::shared_ptr<*ImagePyramid*> **SharedPtr**

typedef *std*::unique_ptr<*ImagePyramid*> **UniquePtr**

Public Functions

inline **ImagePyramid**(int64_t timestamp_, const cv::Mat &image, const cv::Size &winSize, int maxPyrLevel)

Construct the image pyramid.

Parameters

- **timestamp_** – Image timestamp.
- **image** – Image values.
- **winSize** – Window size for the search.
- **maxPyrLevel** – Maximum pyramid layer id (zero-based).

inline **ImagePyramid**(const *dv*::*Frame* &frame, const cv::Size &winSize, int maxPyrLevel)

Construct the image pyramid.

Parameters

- **frame** – *dv::Frame* containing an image and timestamp.
- **winSize** – Window size for the search.
- **maxPyrLevel** – Maximum pyramid layer id (zero-based).

inline **ImagePyramid**(int64_t timestamp_, const cv::Mat &image)

Create a single layer image representation (no pyramid is going to be built).

Parameters

- **timestamp_** – Image timestamp.
- **image** – Image values.

Public Members

int64_t **timestamp**

Timestamp of the image pyramid.

std::vector<cv::Mat> **pyramid**

Pyramid layers of the image.

struct **IMU** : public *flatbuffers*::NativeTable

Public Types

typedef *IMUFlatbuffer* **TableType**

Public Functions

inline **IMU**()

inline **IMU**(int64_t _timestamp, float _temperature, float _accelerometerX, float _accelerometerY, float _accelerometerZ, float _gyroscopeX, float _gyroscopeY, float _gyroscopeZ, float _magnetometerX, float _magnetometerY, float _magnetometerZ)

inline Eigen::Vector3f **getAccelerations**() const

Get measured acceleration in m/s^2.

Returns
Measured acceleration.

inline Eigen::Vector3f **getAngularVelocities**() const

Get measured angular velocities in rad/s.

Returns
Measured angular velocities.

Public Members

int64_t **timestamp**

float **temperature**

float **accelerometerX**

float **accelerometerY**

float **accelerometerZ**

float **gyroscopeX**

float **gyroscopeY**

float **gyroscopeZ**

float **magnetometerX**

float **magnetometerY**

float **magnetometerZ**

Public Static Functions

static inline constexpr const char *__GetFullyQualifiedName__()

struct __IMUBuilder__

Public Functions

inline void __add_timestamp__(int64_t timestamp)

inline void __add_temperature__(float temperature)

inline void __add_accelerometerX__(float accelerometerX)

inline void __add_accelerometerY__(float accelerometerY)

inline void __add_accelerometerZ__(float accelerometerZ)

inline void __add_gyroscopeX__(float gyroscopeX)

inline void __add_gyroscopeY__(float gyroscopeY)

inline void __add_gyroscopeZ__(float gyroscopeZ)

inline void __add_magnetometerX__(float magnetometerX)

inline void __add_magnetometerY__(float magnetometerY)

inline void __add_magnetometerZ__(float magnetometerZ)

inline explicit __IMUBuilder__(*flatbuffers*::FlatBufferBuilder &_fbb)

IMUBuilder &__operator=__(const *IMUBuilder*&)

inline *flatbuffers*::Offset<*IMUFlatbuffer*> __Finish__()

Public Members

flatbuffers::FlatBufferBuilder &__fbb___

flatbuffers::uoffset_t __start___

struct __IMUCalibration__

Public Functions

IMUCalibration() = default

inline **IMUCalibration**(const *std*::string &name, const float omegaMax, const float accMax, const cv::Point3f &omegaOffsetAvg, const cv::Point3f &accOffsetAvg, const float omegaOffsetVar, const float accOffsetVar, const float omegaNoiseDensity, const float accNoiseDensity, const float omegaNoiseRandomWalk, const float accNoiseRandomWalk, const int64_t timeOffsetMicros, *std*::span<const float> transformationToC0View, const *std*::optional<*Metadata*> &metadata)

inline explicit **IMUCalibration**(const pt::ptree &tree)

inline pt::ptree **toPropertyTree**() const

inline bool **operator==**(const *IMUCalibration* &rhs) const

Public Members

std::string **name**

Sensor name (e.g. "IMU_DVXplorer_DXA02137")

float **omegaMax** = -1.f

Maximum (saturation) angular velocity of the gyroscope [rad/s].

float **accMax** = -1.f

Maximum (saturation) acceleration of the accelerometer [m/s^2].

cv::Point3f **omegaOffsetAvg**

Average offset (bias) of the angular velocity [rad/s].

cv::Point3f **accOffsetAvg**

Average offset (bias) of the acceleration [m/s^2].

float **omegaOffsetVar** = -1.f

Variance of the offset of the angular velocity [rad/s].

float **accOffsetVar** = -1.f

Variance of the offset of the acceleration [m/s^2].

float **omegaNoiseDensity** = -1.f

Noise density of the gyroscope [rad/s^s/sqrt(Hz)].

float **accNoiseDensity** = -1.f

Noise density of the accelerometer [m/s^2/sqrt(Hz)].

float **omegaNoiseRandomWalk** = -1.f

Noise random walk of the gyroscope [rad/s^s/sqrt(Hz)].

float **accNoiseRandomWalk** = -1.f

Noise random walk of the accelerometer [m/s^2/sqrt(Hz)].

int64_t **timeOffsetMicros** = -1

Offset between the camera and *IMU* timestamps in microseconds (t_correct = t_imu - offset)

std::vector<float> **transformationToC0**

Transformation converting points in *IMU* frame to C0 frame p_C0= T * p_IMU.

std::optional<*Metadata*> **metadata**

Metadata.

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *IMUCalibration* &calibration)

struct **IMUFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *IMU* **NativeTableType**

Public Functions

inline int64_t **timestamp**() const

Timestamp (µs).

inline float **temperature**() const

Temperature, measured in °C.

inline float **accelerometerX**() const

Acceleration in the X axis, measured in g (9.81m/s^2).

inline float **accelerometerY**() const

Acceleration in the Y axis, measured in g (9.81m/s^2).

inline float **accelerometerZ**() const

Acceleration in the Z axis, measured in g (9.81m/s^2).

inline float **gyroscopeX**() const

Rotation in the X axis, measured in °/s.

inline float **gyroscopeY**() const

Rotation in the Y axis, measured in °/s.

inline float **gyroscopeZ**() const

Rotation in the Z axis, measured in °/s.

inline float **magnetometerX**() const

Magnetometer X axis, measured in µT (magnetic flux density).

inline float **magnetometerY**() const

Magnetometer Y axis, measured in µT (magnetic flux density).

inline float **magnetometerZ**() const

Magnetometer Z axis, measured in µT (magnetic flux density).

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *IMU* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*IMU* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*IMU* *_o, const *IMUFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*IMUFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *IMU* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

struct **IMUPacket** : public *flatbuffers*::NativeTable

Public Types

typedef *IMUPacketFlatbuffer* **TableType**

Public Functions

inline **IMUPacket**()

inline **IMUPacket**(const *dv*::*cvector*<*IMU*> &_elements)

Public Members

dv::*cvector*<*IMU*> **elements**

Public Static Functions

static inline constexpr const char *__GetFullyQualifiedName__()

Friends

inline friend *std*::ostream &__operator<<__(*std*::ostream &os, const *IMUPacket* &packet)

struct __IMUPacketBuilder__

Public Functions

inline void __add_elements__(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*IMUFlatbuffer*>>> elements)

inline explicit __IMUPacketBuilder__(*flatbuffers*::FlatBufferBuilder &_fbb)

IMUPacketBuilder &__operator=__(const *IMUPacketBuilder*&)

inline *flatbuffers*::Offset<*IMUPacketFlatbuffer*> __Finish__()

Public Members

flatbuffers::FlatBufferBuilder &__fbb___

flatbuffers::uoffset_t __start___

struct __IMUPacketFlatbuffer__ : private *flatbuffers*::Table

Public Types

typedef *IMUPacket* __NativeTableType__

Public Functions

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*IMUFlatbuffer*>> *__elements__() const

inline bool __Verify__(*flatbuffers*::Verifier &verifier) const

inline *IMUPacket* *__UnPack__(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void __UnPackTo__(*IMUPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable***()

static inline constexpr const char ***GetFullyQualifiedName***()

static inline void **UnPackToFrom**(*IMUPacket* *_o, const *IMUPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*IMUPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *IMUPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier*** = "IMUS"

struct **Info**

Public Members

bool **imageCompensated** = false

bool **depthAvailable** = false

bool **transformsAvailable** = false

int64_t **depthTime** = -1LL

int64_t **generationTime** = -1LL

size_t **inputEventCount** = 0ULL

size_t **accumulatedEventCount** = 0ULL

struct **InputError**

Public Types

using **Info** = *ErrorInfo*

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

template<class **TYPE**>

struct **InvalidArgument**

Public Types

using **Info** = *TYPE*

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

class **IODataBuffer**

Public Functions

IODataBuffer() = default

inline *dv*::PacketHeader ***getHeader**()

inline const *dv*::PacketHeader ***getHeader**() const

inline *flatbuffers*::FlatBufferBuilder ***getBuilder**()

inline *std*::vector<*std*::byte> ***getBuffer**()

inline const *std*::byte ***getData**() const

inline size_t **getDataSize**() const

inline void **switchToBuffer**()

Private Members

dv::PacketHeader **mHeader**

std::vector<*std*::byte> **mBuffer**

flatbuffers::FlatBufferBuilder **mBuilder** = {*INITIAL_SIZE*}

bool **mIsFlatBuffer** = {true}

Private Static Attributes

static constexpr size_t **INITIAL_SIZE** = {64 * 1024}

struct **IOError** : public *dv*::*exceptions*::*info*::*EmptyException*

struct **IOHeader** : public *flatbuffers*::NativeTable

Public Types

typedef *IOHeaderFlatbuffer* **TableType**

Public Functions

inline **IOHeader**()

inline **IOHeader**(*CompressionType* _compression, int64_t _dataTablePosition, const *dv*::*cstring* &_infoNode)

Public Members

CompressionType **compression**

int64_t **dataTablePosition**

dv::*cstring* **infoNode**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

struct **IOHeaderBuilder**

Public Functions

inline void **add_compression**(*CompressionType* compression)

inline void **add_dataTablePosition**(int64_t dataTablePosition)

inline void **add_infoNode**(*flatbuffers*::Offset<*flatbuffers*::String> infoNode)

inline explicit **IOHeaderBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

IOHeaderBuilder &**operator=**(const *IOHeaderBuilder*&)

inline *flatbuffers*::Offset<*IOHeaderFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **IOHeaderFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *IOHeader* **NativeTableType**

Public Functions

inline *CompressionType* **compression**() const

inline int64_t **dataTablePosition**() const

inline const *flatbuffers*::String ***infoNode**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *IOHeader* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*IOHeader* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*IOHeader* *_o, const *IOHeaderFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*IOHeaderFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *IOHeader* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "IOHE"

class **IOStatistics**

Public Functions

IOStatistics() = default

virtual **~IOStatistics**() = default

IOStatistics(const *IOStatistics* &other) = delete

IOStatistics &**operator=**(const *IOStatistics* &other) = delete

IOStatistics(*IOStatistics* &&other) noexcept = default

IOStatistics &**operator=**(*IOStatistics* &&other) = default

virtual void **publish**() = 0

inline void **addBytes**(const uint64_t bytes)

inline void **update**(const uint64_t addedDataSize, const uint64_t addedPacketsNumber, const uint64_t addedPacketsElements, const uint64_t addedPacketsSize)

Protected Attributes

uint64_t **mPacketsNumber** = {0}

uint64_t **mPacketsElements** = {0}

uint64_t **mPacketsSize** = {0}

uint64_t **mDataSize** = {0}

template<typename **T**>

struct **is_eigen_impl** : public *std*::false_type

template<typename **T**, int... **Is**>

struct **is_eigen_impl**<Eigen::Matrix<*T*, *Is*...>> : public *std*::true_type

class **KDTreeEventStoreAdaptor**

#include *</builds/inivation/dv/dv-processing/include/dv-processing/containers/kd_tree/event_store_adaptor.hpp>*
Wrapper class around nanoflann::KDTree for dv::EventStore data, which provides efficient approximate nearest neighbour search as well as radius search.

Public Functions

inline **KDTreeEventStoreAdaptor**(const *dv*::*EventStore* &data, const uint32_t maxLeaves = 32768)

Constructor

See also:

MeanShift::Matrix

Template Parameters

T – The matrix type. Must be of exact same type as MeanShift::Matrix, to avoid copy construction of a temporary variable and thereby creating dangling references.

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **maxLeaves** – the maximum number of leaves for the KDTree. A smaller number typically increases the time used for construction of the tree, but may decrease the time used for searching it. A higher number typically does the opposite.

KDTreeEventStoreAdaptor() = delete

KDTreeEventStoreAdaptor(const *KDTreeEventStoreAdaptor* &other) = delete

KDTreeEventStoreAdaptor(*KDTreeEventStoreAdaptor* &&other) = delete

KDTreeEventStoreAdaptor &**operator=**(const *KDTreeEventStoreAdaptor* &other) = delete

KDTreeEventStoreAdaptor &**operator=**(*KDTreeEventStoreAdaptor* &&other) = delete

~KDTreeEventStoreAdaptor() = default

template<typename **T**>
inline auto **knnSearch**(const cv::Point_<*T*> &centrePoint, const size_t numClosest) const

Searches for the k nearest neighbours surrounding centrePoint.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **numClosest** – The number of neighbours to be searched (i.e. the parameter “k”)

Returns

The number of actually found neighbours

inline auto **knnSearch**(const *dv*::Event &centrePoint, const size_t numClosest) const

Searches for the k nearest neighbours surrounding centrePoint.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **numClosest** – The number of neighbours to be searched (i.e. the parameter “k”)

Returns

The number of actually found neighbours

inline auto **knnSearch**(const *dv*::*TimedKeyPoint* &centrePoint, const size_t numClosest) const

Searches for the k nearest neighbours surrounding centrePoint.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **numClosest** – The number of neighbours to be searched (i.e. the parameter "k")

Returns

The number of actually found neighbours

inline *std*::vector<*std*::pair<const *dv*::Event*, int32_t>> **knnSearch**(const int32_t x, const int32_t y, const size_t numClosest) const

Searches for the k nearest neighbours surrounding centrePoint.

Parameters

- **x** – The x-coordinate of the centre point for which the nearest neighbours are to be searched
- **y** – The y-coordinate of the centre point for which the nearest neighbours are to be searched
- **numClosest** – The number of neighbours to be searched (i.e. the parameter "k")

Returns

The number of actually found neighbours

template<typename **T**>
inline auto **radiusSearch**(const cv::Point_<*T*> &centrePoint, const int16_t &radius, float eps = 0.0f, bool sorted = false) const

Searches for all neighbours surrounding centrePoint that are within a certain radius.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **radius** – The radius
- **eps** – The search accuracy
- **sorted** – True if the neighbours should be sorted with respect to their distance to centrePoint (comes with a significant performance impact)

Returns

The number of actually found neighbours

inline auto **radiusSearch**(const *dv*::Event &centrePoint, const int16_t &radius, float eps = 0.0f, bool sorted = false) const

Searches for all neighbours surrounding centrePoint that are within a certain radius.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **radius** – The radius
- **eps** – The search accuracy
- **sorted** – True if the neighbours should be sorted with respect to their distance to centrePoint (comes with a significant performance impact)

Returns

The number of actually found neighbours

inline auto **radiusSearch**(const *dv*::*TimedKeyPoint* &centrePoint, const int16_t &radius, float eps = 0.0f, bool sorted = false) const

Searches for all neighbours surrounding centrePoint that are within a certain radius.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **radius** – The radius
- **eps** – The search accuracy
- **sorted** – True if the neighbours should be sorted with respect to their distance to centrePoint (comes with a significant performance impact)

Returns
The number of actually found neighbours

inline *std*::vector<*std*::pair<const *dv*::Event*, int32_t>> **radiusSearch**(const int32_t x, int32_t y, const int16_t &radius, float eps = 0.0f, bool sorted = false) const

Searches for all neighbours surrounding centrePoint that are within a certain radius.

Parameters

- **x** – The x-coordinate of the centre point for which the nearest neighbours are to be searched
- **y** – The y-coordinate of the centre point for which the nearest neighbours are to be searched
- **radius** – The radius
- **eps** – The search accuracy
- **sorted** – True if the neighbours should be sorted with respect to their distance to centrePoint (comes with a significant performance impact)

Returns
The number of actually found neighbours

inline *dv*::*EventStore*::iterator **begin**() const noexcept

Returns an iterator to the begin of the EventStore

Returns
an iterator to the begin of the EventStore

inline *dv*::*EventStore*::iterator **end**() const noexcept

Returns an iterator to the end of the EventStore

Returns
an iterator to the end of the EventStore

inline const *KDTreeEventStoreAdaptor* &**derived**() const

Returns the reference to the this object. Required by the nanoflann adaptors

Returns
the reference to “this”

inline *KDTreeEventStoreAdaptor* &**derived**()

Returns the reference to the this object. Required by the nanoflann adaptors

Returns
the reference to “this”

inline uint32_t **kdtree_get_point_count**() const

Returns the point count of the event store. Required by the nanoflann adaptors

Returns
the reference to "this"

inline int16_t **kdtree_get_pt**(const *dv*::Event *event, const size_t dim) const

Returns the dim'th dimension of an event. Required by the nanoflann adaptors

Returns
the reference to "this"

template<class **BBOX**>
inline bool **kdtree_get_bbox**(*BBOX*&) const

Bounding box computation required by the nanoflann adaptors As the documentation allows for it not being implemented and we don't need it, it was left empty.

Returns
false

Private Types

using **Index** =
nanoflann::KDTreeSingleIndexNonContiguousIteratorAdaptor<nanoflann::metric_L2_Simple::traits<int32_t, *KDTreeEventStoreAdaptor*, const *dv*::Event*>::distance_t, *KDTreeEventStoreAdaptor*, 2, const *dv*::Event*>

Private Members

const *dv*::*EventStore* &**mData**

std::unique_ptr<*Index*> **mIndex**

template<typename **TYPE**, int32_t **ROWS** = Eigen::Dynamic, int32_t **COLUMNS** = Eigen::Dynamic, int32_t **SAMPLE_ORDER** = Eigen::ColMajor>
class **KDTreeMatrixAdaptor**

#include </builds/inivation/dv/dv-processing/include/dv-processing/containers/kd_tree/eigen_matrix_adaptor.hpp>
Wrapper class around nanoflann::KDTree for data contained in Eigen matrices, which provides efficient approximate nearest neighbour search as well as radius search.

See also:

Eigen::Dynamic

See also:

Eigen::Dynamic

See also:

Eigen::StorageOptions

Template Parameters

- **TYPE** – the underlying data type
- **ROWS** – the number of rows in the data matrix. May be Eigen::Dynamic or >= 0.

- **COLUMNS** – the number of columns in the data matrix. May be Eigen::Dynamic or >= 0.
- **SAMPLE_ORDER** – the order in which samples are entered in the matrix.

Public Types

using **Matrix** = Eigen::Matrix<*TYPE*, *ROWS*, *COLUMNS*, *STORAGE_ORDER*>

using **Vector** = Eigen::Matrix<*TYPE*, *SAMPLE_ORDER* == Eigen::ColMajor ? *ROWS* : 1, *SAMPLE_ORDER* == Eigen::ColMajor ? 1 : *COLUMNS*, *STORAGE_ORDER*>

Public Functions

template<typename **T**, *std*::enable_if_t<*std*::is_same_v<*T*, *Matrix*>, bool> = false>
inline explicit **KDTreeMatrixAdaptor**(const *T* &data, const uint32_t maxLeaves = 32768)

Constructor

See also:

MeanShift::Matrix

Template Parameters

T – The matrix type. Must be of exact same type as MeanShift::Matrix, to avoid copy construction of a temporary variable and thereby creating dangling references.

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **maxLeaves** – the maximum number of leaves for the KDTree. A smaller number typically increases the time used for construction of the tree, but may decrease the time used for searching it. A higher number typically does the opposite.

KDTreeMatrixAdaptor() = delete

KDTreeMatrixAdaptor(const *ThisType* &other) = delete

KDTreeMatrixAdaptor(*ThisType* &&other) = delete

KDTreeMatrixAdaptor &**operator=**(const *ThisType* &other) = delete

KDTreeMatrixAdaptor &**operator=**(*ThisType* &&other) = delete

~KDTreeMatrixAdaptor() = default

inline auto **knnSearch**(const *Vector* &centrePoint, const size_t numClosest) const

Searches for the k nearest neighbours surrounding centrePoint.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **numClosest** – The number of neighbours to be searched (i.e. the parameter “k”)

Returns
A pair containing the indices of the neighbours in the underlying matrix as well as the distances to centrePoint

inline auto **radiusSearch**(const *Vector* &centrePoint, const *TYPE* &radius, float eps = 0.0f, bool sorted = false) const

Searches for all neighbours surrounding centrePoint that are within a certain radius.

Parameters

- **centrePoint** – The point for which the nearest neighbours are to be searched
- **radius** – The radius
- **eps** – The search accuracy
- **sorted** – True if the neighbours should be sorted with respect to their distance to centrePoint (comes with a significant performance impact)

Returns
A vector of pairs containing the indices of the neighbours in the underlying matrix as well as the distances to centrePoint

inline auto **getSample**(const uint32_t index) const

Returns a sample at a given index

Parameters
index – the index of the sample in mData

Returns
the sample

Private Types

using **ThisType** = *KDTreeMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, *SAMPLE_ORDER*>

using **Tree** = nanoflann::KDTreeEigenMatrixAdaptor<*Matrix*, *SAMPLE_ORDER* == Eigen::ColMajor ? *ROWS* : *COLUMNS*, nanoflann::metric_L2_Simple, *SAMPLE_ORDER* == Eigen::RowMajor>

Private Members

const uint32_t **mNumSamples**

const uint32_t **mNumDimensions**

std::unique_ptr<*Tree*> **mTree**

Private Static Attributes

static constexpr int32_t **DIMS** = *SAMPLE_ORDER* == Eigen::ColMajor ? *ROWS* : *COLUMNS*

static constexpr int32_t **NOT_SAMPLE_ORDER** = (*SAMPLE_ORDER* == Eigen::ColMajor ? Eigen::RowMajor : Eigen::ColMajor)

static constexpr int32_t **STORAGE_ORDER** = *DIMS* == 1 ? *NOT_SAMPLE_ORDER* : *SAMPLE_ORDER*

class **KeyPointResampler**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/keypoint_resampler.hpp> Create a feature resampler, which resamples given keypoints with homogenous distribution in pixel space.

Implementation was inspired by: https://github.com/BAILOOL/ANMS-Codes

Public Functions

inline explicit **KeyPointResampler**(const cv::Size &resolution)

Initialize resampler with given resolution.

Parameters
resolution – Image resolution

template<class KeyPointVectorType> inline or dv::concepts::Coordinate2DMutableIterable< KeyPointVectorType > KeyPointVectorType resample (const KeyPointVectorType &keyPoints, size_t numRetPoints)

Perform resampling on given keypoints.

See also:

setTolerance)

Parameters

- **keyPoints** – Prior keypoints.
- **numRetPoints** – Number of expected keypoints, the exact number of output keypoints can vary to configured tolerance value (

Returns
Resampled keypoints

inline float **getTolerance**() const

Get currently set tolerance for output keypoint count.

Returns
Tolerance value

inline void **setTolerance**(const float tolerance)

Set a new output size tolerance value.

The algorithm search for an optimal distance between keypoints so the resulting vector would contain the expected amount of keypoints. This search is performed with a given tolerance, by default - 0.1 (so by default the final resampled amount of events will be within +/-10% of requested amount).

Parameters
tolerance – Output keypoint amount tolerance value.

Protected Types

typedef *std*::pair<*dv*::Point2f, size_t> **RangeValue**

Protected Attributes

float **mPreviousSolution** = -1.f

float **mRows**

float **mCols**

float **mTolerance** = 0.1f

struct **Landmark** : public *flatbuffers*::NativeTable

Public Types

typedef *LandmarkFlatbuffer* **TableType**

Public Functions

inline **Landmark**()

inline **Landmark**(const Point3f &_pt, int64_t _id, int64_t _timestamp, const *dv*::*cvector*<int8_t> &_descriptor, const *dv*::*cstring* &_descriptorType, const *dv*::*cvector*<float> &_covariance, const *dv*::*cvector*<*Observation*> &_observations)

Public Members

Point3f **pt**

int64_t **id**

int64_t **timestamp**

dv::*cvector*<int8_t> **descriptor**

dv::*cstring* **descriptorType**

dv::*cvector*<float> **covariance**

dv::*cvector*<*Observation*> **observations**

Public Static Functions

static inline constexpr const char *__GetFullyQualifiedName__()

struct **LandmarkBuilder**

Public Functions

inline void **add_pt**(const Point3f *pt)

inline void **add_id**(int64_t id)

inline void **add_timestamp**(int64_t timestamp)

inline void **add_descriptor**(*flatbuffers*::Offset<*flatbuffers*::Vector<int8_t>> descriptor)

inline void **add_descriptorType**(*flatbuffers*::Offset<*flatbuffers*::String> descriptorType)

inline void **add_covariance**(*flatbuffers*::Offset<*flatbuffers*::Vector<float>> covariance)

inline void **add_observations**(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*ObservationFlatbuffer*>>> observations)

inline explicit **LandmarkBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

LandmarkBuilder &**operator=**(const *LandmarkBuilder*&)

inline *flatbuffers*::Offset<*LandmarkFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **LandmarkFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *Landmark* **NativeTableType**

Public Functions

inline const Point3f *__pt__() const

3D coordinate of the landmark.

inline int64_t __id__() const

Landmark id (if the keypoints need to be clustered by an object they belong to).

inline int64_t __timestamp__() const

Timestamp (µs).

inline const *flatbuffers*::Vector<int8_t> *__descriptor__() const

Visual descriptor of the landmark.

inline const *flatbuffers*::String *__descriptorType__() const

Type of the visual descriptor.

inline const *flatbuffers*::Vector<float> *__covariance__() const

Covariance matrix, must contain 9 numbers. It is represented as a 3x3 square matrix.

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*ObservationFlatbuffer*>> *__observations__() const

Observation info, can be from multiple cameras if they are matched using descriptor.

inline bool __Verify__(*flatbuffers*::Verifier &verifier) const

inline *Landmark* *__UnPack__(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void __UnPackTo__(*Landmark* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable *__MiniReflectTypeTable__()

static inline constexpr const char *__GetFullyQualifiedName__()

static inline void __UnPackToFrom__(*Landmark* *_o, const *LandmarkFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*LandmarkFlatbuffer*> __Pack__(*flatbuffers*::FlatBufferBuilder &_fbb, const *Landmark* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

struct **LandmarksPacket** : public *flatbuffers*::NativeTable

Public Types

typedef *LandmarksPacketFlatbuffer* **TableType**

Public Functions

inline **LandmarksPacket**()

inline **LandmarksPacket**(const *dv*::*cvector*<*Landmark*> &_elements, const *dv*::*cstring* &_referenceFrame)

Public Members

dv::*cvector*<*Landmark*> **elements**

dv::*cstring* **referenceFrame**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *LandmarksPacket* &packet)

struct **LandmarksPacketBuilder**

Public Functions

inline void **add_elements**(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*LandmarkFlatbuffer*>>> elements)

inline void **add_referenceFrame**(*flatbuffers*::Offset<*flatbuffers*::String> referenceFrame)

inline explicit **LandmarksPacketBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

LandmarksPacketBuilder &**operator=**(const *LandmarksPacketBuilder*&)

inline *flatbuffers*::Offset<*LandmarksPacketFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **LandmarksPacketFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *LandmarksPacket* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*LandmarkFlatbuffer*>> ***elements**() const

inline const *flatbuffers*::String ***referenceFrame**() const

Coordinate reference frame of the landmarks, "world" coordinate frame by default.

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *LandmarksPacket* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*LandmarksPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*LandmarksPacket* *_o, const *LandmarksPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*LandmarksPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *LandmarksPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "LMRS"

struct **LengthError** : public *dv*::*exceptions*::*info*::*EmptyException*

template<*std*::floating_point **Scalar**>

class **LinearTransformer**

#include </builds/inivation/dv/dv-processing/include/dv-processing/kinematics/linear_transformer.hpp> A buffer containing time increasing 3D transformations and capable of timewise linear interpolation between available transforms. Can be used with different underlying floating point types supported by Eigen.

Template Parameters
Scalar – Underlying floating point number type - float or double.

Public Types

using **iterator** = typename *TransformationBuffer*::iterator

using **const_iterator** = typename *TransformationBuffer*::const_iterator

Public Functions

inline explicit **LinearTransformer**(size_t capacity)

inline void **pushTransformation**(const *TransformationType* &transformation)

Push a transformation into the transformation buffer.

Throws
`logic_error` – exception when transformation is added out of order.

Parameters
transformation – *Transformation* to be pushed, it must contain increasing timestamp compared to latest transformation in the buffer, otherwise an exception will be thrown.

inline *iterator* **begin**()

Generate forward iterator pointing to first transformation in the transformer buffer.

Returns
Buffer start iterator.

inline *iterator* **end**()

Generate an iterator representing end of the buffer.

Returns
Buffer end const-iterator.

inline *const_iterator* **cbegin**() const

Generate a const forward iterator pointing to first transformation in the transformer buffer.

Returns
Buffer start const-iterator.

inline *const_iterator* **cend**() const

Generate a const iterator representing end of the buffer.

Returns
Buffer end iterator.

inline void **clear**()

Delete all transformations from the buffer.

inline bool **empty**() const

Check whether the buffer is empty.

Returns
true if empty, false otherwise

inline *std*::optional<*TransformationType*> **getTransformAt**(int64_t timestamp) const

Get a transform at the given timestamp.

If no transform with the exact timestamp was pushed, estimates a transform assuming linear motion.

Parameters
timestamp – Unix timestamp in microsecond format.

Returns
Transformation if successful, std::nullopt otherwise.

inline bool **isWithinTimeRange**(int64_t timestamp) const

Checks whether the timestamp is within the range of transformations available in the buffer.

Parameters
timestamp – Unix microsecond timestamp to be checked.

Returns
true if the timestamp is within the range of transformations in the buffer.

inline size_t **size**() const

Return the size of the buffer.

Returns
Number of transformations available in the buffer.

inline const *TransformationType* &**latestTransformation**() const

Return transformation with highest timestamp.

Returns
Latest transformation in the buffer.

inline const *TransformationType* &**earliestTransformation**() const

Return transformation with lowest timestamp.

Returns
Earliest transformation in time available in the buffer.

inline void **setCapacity**(size_t newCapacity)

Set new capacity, if the size of the buffer is larger than the newCapacity, oldest transformations from the start will be removed.

Parameters
newCapacity – New transformation buffer capacity.

inline *LinearTransformer*<*Scalar*> **getTransformsBetween**(int64_t start, int64_t end) const

Extract transformation between two given timestamps. If timestamps are not at exact available transformations, additional transformations will be added so the resulting transformer would complete overlap over the period (if that is possible).

Parameters

- **start** – Start Unix timestamp in microseconds.
- **end** – End Unix timestamp in microseconds.

Returns
LinearTransformer containing transformations covering the given period.

inline *LinearTransformer*<*Scalar*> **resampleTransforms**(const int64_t samplingInterval) const

Resample containing transforms into a new transformer, containing interpolated transforms at given interval. Will contain the last transformation as well, although the interval might not be maintained for the last transform.

Parameters
samplingInterval – Interval in microseconds at which to resample the transformations.

Returns
Generated transformer with exact capacity of output transformation count.

Private Types

using **TransformationType** = *Transformation*<*Scalar*>

using **TransformationBuffer** = boost::circular_buffer<*TransformationType*, Eigen::aligned_allocator<*TransformationType*>>

Private Functions

inline *TransformationBuffer*::const_iterator **bufferLowerBound**(int64_t t) const

Finds the lower bound iterator in the buffer.

See also:

std::lower_bound

Parameters
t – Unix timestamp in microseconds to search for.

Returns
Iterator to the buffer with timestamp that is *equal or not less* than given timestamp.

inline *TransformationBuffer*::const_iterator **bufferUpperBound**(int64_t t) const

Finds the upper bound iterator in the buffer.

See also:

std::upper_bound

Parameters
t – Unix timestamp in microseconds to search for.

Returns
Iterator to the buffer with timestamp that is *greater* than given timestamp or `end` if not available.

Private Members

TransformationBuffer **mTransforms**

Private Static Functions

static inline *TransformationType* **interpolateComponentwise**(const *TransformationType* &T_a, const *TransformationType* &T_b, const int64_t timestamp, *Scalar* lambda)

Perform linear interpolation between two transformations.

Parameters

- **T_a** – First transformation.
- **T_b** – Second transformation.
- **timestamp** – Interpolated transformation timestamp.
- **lambda** – Distance point between the two transformation to interpolate.

Returns
Interpolated transformation.

struct **LucasKanadeConfig**

#include *</builds/inivation/dv/dv-processing/include/dv-processing/features/image_feature_lk_tracker.hpp>*
Lucas-Kanade tracker configuration parameters.

Public Members

bool **maskedFeatureDetect** = true

Generate a mask which would disable image regions where features are already succesfully tracked.

double **terminationEpsilon** = 0.1

Tracking termination criteria for the LK tracker.

int **numPyrLayers** = 2

Total number of pyramid layers used by the LK tracker.

cv::Size **searchWindowSize** = cv::Size(24, 24)

Size of the search around the tracked feature.

class **Lz4CompressionSupport** : public *dv*::*io*::*compression*::*CompressionSupport*

Public Functions

inline explicit **Lz4CompressionSupport**(const *CompressionType* type)

inline explicit **Lz4CompressionSupport**(const LZ4F_preferences_t &preferences)

LZ4 compression support with custom compression settings. Internally sets compression type to `CompressionType::LZ4`.

Parameters
preferences – LZ4 compression settings.

inline virtual void **compress**(*dv*::*io*::*support*::*IODataBuffer* &packet) override

Private Members

std::shared_ptr<LZ4F_cctx_s> **mContext**

const LZ4F_preferences_t **mPrefs**

size_t **mChunkSize**

size_t **mEndSize**

Private Static Attributes

static constexpr size_t **LZ4_COMPRESSION_CHUNK_SIZE** = {64 * 1024}

static constexpr LZ4F_preferences_t **lz4CompressionPreferences** = {{LZ4F_max64KB, LZ4F_blockLinked, LZ4F_noContentChecksum, LZ4F_frame}, 0, 0,}

static constexpr LZ4F_preferences_t **lz4HighCompressionPreferences** = {{LZ4F_max64KB, LZ4F_blockLinked, LZ4F_noContentChecksum, LZ4F_frame}, 9, 0,}

class **Lz4DecompressionSupport** : public *dv*::*io*::*compression*::*DecompressionSupport*

Public Functions

inline explicit **Lz4DecompressionSupport**(const *CompressionType* type)

inline virtual void **decompress**(*std*::vector<*std*::byte> &src, *std*::vector<*std*::byte> &target) override

Private Functions

inline void **initDecompressionContext**()

Private Members

std::shared_ptr<LZ4F_dctx_s> **mContext**

Private Static Attributes

static constexpr size_t **LZ4_DECOMPRESSION_CHUNK_SIZE** = {64 * 1024}

class **MapOfVariants** : public *std*::unordered_map<*std*::string, InputType>

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/multi_stream_slicer.hpp> Class that is passed to the slicer callback. It is an unordered map where key is the configured stream name and the value is a variant. The class provides convenience methods to access and cast the types.

Public Functions

template<class **Type**>
inline *Type* &**get**(const *std*::string &streamName)

Get a reference to the data packet of a given stream name.

Template Parameters
Type – Type of data for the stream.

Parameters
streamName – Stream name.

Returns
Data packet casted to the given type.

template<class **Type**>
inline const *Type* &**get**(const *std*::string &streamName) const

Get a reference to the data packet of a given stream name.

Template Parameters
Type – Type of data for the stream.

Parameters
streamName – Stream name.

Returns
Data packet casted to the given type.

struct **Marker**

Public Functions

inline **Marker**(int64_t timestamp, bool active, const Eigen::Vector3f &point)

Public Members

EIGEN_MAKE_ALIGNED_OPERATOR_NEW int64_t timestamp

bool **active**

Eigen::Vector3f **point**

template<typename **TYPE**, int32_t **ROWS** = Eigen::Dynamic, int32_t **COLUMNS** = Eigen::Dynamic, int32_t **SAMPLE_ORDER** = Eigen::ColMajor>
class **MeanShiftEigenMatrixAdaptor**

#include </builds/inivation/dv/dv-processing/include/dv-processing/cluster/mean_shift/eigen_matrix_adaptor.hpp>
This class implements the Mean Shift clustering algorithm.

As the Mean Shift algorithm performs a gradient ascent on an estimated probability density function, when applying it to integer data, which has a non-smooth probability density, the quality of the detected clusters depends significantly on the selected bandwidth hyperparameter, as well as the underlying data and the selected kernel. Generally the Gaussian Kernel yields better results for this kind of data, however it comes with a bigger performance impact.

The Mean Shift algorithm is an nonparametric estimate of the modes of the underlying probability distribution for the data. It implements an iterative search, starting from points provided by the user, or randomly selected from the data points provided. For each iteration, the current estimate of the mode is replaced by an estimate of the mean value of the surrounding data samples. If the Epanechnikov kernel is used for the underlying density estimate, its so-called "shadow kernel", the flat kernel must be used for the estimate of the mean. This means, that we can simply compute the average value of the data points that lie within a given radius around the current estimate of the mode, and use this as the next estimate. To provide an efficient search for the neighbours of the current mode estimate, a KD tree was used.

For the underlying theory, see "The Estimation of the Gradient of a Density Function with

Applications in Pattern Recognition" by K. Fukunaga and L. Hostetler as well as "Mean shift, mode seeking, and

clustering" by Yizong Cheng.

See also:

Eigen::Dynamic

See also:

Eigen::Dynamic

See also:

Eigen::StorageOptions

Template Parameters

- **TYPE** – the underlying data type
- **ROWS** – the number of rows in the data matrix. May be Eigen::Dynamic or >= 0.
- **COLUMNS** – the number of columns in the data matrix. May be Eigen::Dynamic or >= 0.
- **SAMPLE_ORDER** – the order in which samples are entered in the matrix.

Public Types

using **Matrix** = Eigen::Matrix<*TYPE*, *ROWS*, *COLUMNS*, *STORAGE_ORDER*>

using **Vector** = Eigen::Matrix<*TYPE*, *SAMPLE_ORDER* == Eigen::ColMajor ? *ROWS* : 1, *SAMPLE_ORDER* == Eigen::ColMajor ? 1 : *COLUMNS*, *STORAGE_ORDER*>

using **VectorOfVectors** = *std*::vector<*Vector*, Eigen::aligned_allocator<*Vector*>>

Public Functions

template<typename **T**, *std*::enable_if_t<*std*::is_same_v<*T*, *Matrix*>, bool> = false>
inline **MeanShiftEigenMatrixAdaptor**(const *T* &data, const *TYPE* bw, *TYPE* conv, const uint32_t maxIter, const *VectorOfVectors* &startingPoints, const uint32_t numLeaves = 32768)

Constructor

See also:

MeanShift::Matrix

See also:

dv::containers::KDTree

Template Parameters

T – The matrix type. Must be of exact same type as MeanShift::Matrix, to avoid copy construction of a temporary variable and thereby creating dangling references.

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **bw** – The bandwidth used for the shift. This is a hyperparameter for the kernel. For the Epanechnikov kernel this means that all values within a radius of bw are averaged.
- **conv** – For each starting point, the algorithm is stopped as soon as the absolute value of the shift is <= conv.
- **maxIter** – The maximum number of iterations. Detected modes, for which the the number of iterations exceed this value are not added to the detected clusters.
- **startingPoints** – Points from which to start the search.
- **numLeaves** – the maximum number of leaves for the KDTree.

template<typename **T**, *std*::enable_if_t<*std*::is_same_v<*T*, *Matrix*>, bool> = false>
inline **MeanShiftEigenMatrixAdaptor**(const *T* &data, const *TYPE* bw, *TYPE* conv, const uint32_t maxIter, *VectorOfVectors* &&startingPoints, const uint32_t numLeaves = 32768)

Constructor

See also:

MeanShift::Matrix

See also:

dv::containers::KDTree

Template Parameters

T – The matrix type. Must be of exact same type as MeanShift::Matrix, to avoid copy construction of a temporary variable and thereby creating dangling references.

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **bw** – The bandwidth used for the shift. This is a hyperparameter for the kernel. For the Epanechnikov kernel this means that all values within a radius of bw are averaged.
- **conv** – For each starting point, the algorithm is stopped as soon as the absolute value of the shift is <= conv.
- **maxIter** – The maximum number of iterations. Detected modes, for which the the number of iterations exceed this value are not added to the detected clusters.
- **startingPoints** – Points from which to start the search.
- **numLeaves** – the maximum number of leaves for the KDTree.

template<typename **T**, *std*::enable_if_t<*std*::is_same_v<*T*, *Matrix*>, bool> = false>
inline **MeanShiftEigenMatrixAdaptor**(const *T* &data, const *TYPE* bw, *TYPE* conv, const uint32_t maxIter, const uint32_t numStartingPoints, const uint32_t numLeaves = 32768)

Constructor

See also:

MeanShift::Matrix

See also:

dv::containers::KDTree

Template Parameters

T – The matrix type. Must be of exact same type as MeanShift::Matrix, to avoid copy construction of a temporary variable and thereby creating dangling references.

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **bw** – The bandwidth used for the shift. This is a hyperparameter for the kernel. For the Epanechnikov kernel this means that all values within a radius of bw are averaged.
- **conv** – For each starting point, the algorithm is stopped as soon as the absolute value of the shift is <= conv.
- **maxIter** – The maximum number of iterations. Detected modes, for which the the number of iterations exceed this value are not added to the detected clusters.
- **numStartingPoints** – The number of points which are randomly selected from the data points, to be used as starting points.
- **numLeaves** – the maximum number of leaves for the KDTree.

MeanShiftEigenMatrixAdaptor() = delete

MeanShiftEigenMatrixAdaptor(const *ThisType* &other) = delete

MeanShiftEigenMatrixAdaptor(*ThisType* &&other) = delete

MeanShiftEigenMatrixAdaptor &**operator=**(const *ThisType* &other) = delete

MeanShiftEigenMatrixAdaptor &**operator=**(*ThisType* &&other) = delete

~MeanShiftEigenMatrixAdaptor() = default

template<*kernel*::MeanShiftKernel **kernel** = *kernel*::Epanechnikov>
inline auto **fit**()

Executes the algorithm.

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Returns
The centres of each detected cluster

Returns
The labels for each data point. The labels correspond to the index of the centre to which the sample is assigned.

Returns
The number of samples in each cluster

Returns
The in-cluster variance for each cluster

Public Static Functions

static inline *VectorOfVectors* **generateStartingPointsFromData**(const uint32_t numStartingPoints, const *Matrix* &data)

Generates a vector of vectors containing the starting points by randomly selecting from provided data

Parameters

- **numStartingPoints** – The number of points to be generated
- **data** – the matrix to select the starting points from

Returns
The vector of vectors containing the starting points.

static inline *VectorOfVectors* **generateStartingPointsFromRange**(const uint32_t numStartingPoints, const *std*::vector<*std*::pair<*TYPE*, *TYPE*>> &ranges)

Generates a vector of vectors containing the starting points by generating random points within a given range for each dimension

Parameters

- **numStartingPoints** – The number of points to be generated

- **ranges** – a vector containing one range per dimension. Each dimension is represented by a pair containing the beginning and the end of the range

Returns
The vector of vectors containing the starting points.

Private Types

using **ThisType** = *MeanShiftEigenMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, *SAMPLE_ORDER*>

using **KDTree** = *dv*::*containers*::*kd_tree*::*KDTreeMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, *SAMPLE_ORDER*>

Private Functions

template<*kernel*::MeanShiftKernel **kernel**>
inline auto **findClusterCentres**()

Performs the search for the cluster centres for each given starting point. A detected centre is added to the set of centres if it isn't closer than the bandwidth to any previously detected centre.

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Returns
The centres of each detected cluster

inline auto **assignClusters**(const *VectorOfVectors* &clusterCentres)

Assigns the data samples to a cluster by means of a nearest neighbour search, and computes the number of samples as well as the in-cluster variance in the process.

Parameters
clusterCentres – The centres of each detected cluster

Returns
The labels for each data point. The labels correspond to the index of the centre to which the sample is assigned.

Returns
The number of samples in each cluster

Returns
The in-cluster variance for each cluster

template<*kernel*::MeanShiftKernel **kernel**>
inline *std*::optional<*Vector*> **performShift**(*Vector* currentMode)

Performs a search for a mode in the underlying density starting off with a provided initial point.

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Parameters
currentMode – The starting point that is to be shifted until convergence.

Returns
An std::optional containing either a vector, if the search has converged, std::nullopt otherwise

template<*kernel*::MeanShiftKernel **kernel**>
inline float **applyKernel**(const float squaredDistance) const

Applies the selected kernel to the squared distance

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Parameters
squaredDistance – the squared distance between the current mode estimate and a given sample point

Returns
the kernel value

template<*kernel*::MeanShiftKernel **kernel**>
inline auto **getNeighbours**(const *Vector* &currentMode)

Returns the neighbours surrounding a centre

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Parameters
centre – the centre surrounding which the neighbours are to be found

Returns
the neighbours, as a vector of pairs, one pair per neighbour containing a the index of the point in the data matrix and a distance to the centre

inline auto **getSample**(const uint32_t index) const

Returns a sample at a given index

Parameters
index – the index of the sample in mData

Returns
the sample

inline *Vector* **getZeroVector**() const

Returns
a zero vector of length mNumDimensions

Private Members

const size_t **mNumSamples**

const size_t **mNumDimensions**

KDTree **mData**

const *TYPE* **mBandwidth**

const uint32_t **mMaxIter**

const *TYPE* **mConvergence**

VectorOfVectors **mStartingPoints**

Private Static Functions

template<typename **T**>
static inline auto **randomArrayBetween**(const uint32_t length, const *T* begin, const *T* end)

Generate an array of random values within a given range and a given length

Template Parameters
T – The data type

Parameters

- **length** – The length of the array
- **begin** – The minimum value contained in the array
- **end** – The maximum value contained in the array

Returns
The array

static inline auto **extractSample**(const *Matrix* &data, const uint32_t index)

Returns a sample at a given index

Parameters

- **data** – the data to extract the sample from
- **index** – the index of the sample in mData

Returns
the sample

static inline *Vector* **getZeroVector**(uint32_t numDimensions)

Returns
a zero vector of length mNumDimensions

Private Static Attributes

static constexpr int32_t **DIMS** = *SAMPLE_ORDER* == Eigen::ColMajor ? *ROWS* : *COLUMNS*

static constexpr int32_t **NOT_SAMPLE_ORDER** = (*SAMPLE_ORDER* == Eigen::ColMajor ? Eigen::RowMajor : Eigen::ColMajor)

static constexpr int32_t **STORAGE_ORDER** = *DIMS* == 1 ? *NOT_SAMPLE_ORDER* : *SAMPLE_ORDER*

class **MeanShiftEventStoreAdaptor**

#include </builds/inivation/dv/dv-processing/include/dv-processing/cluster/mean_shift/event_store_adaptor.hpp>
This class implements the Mean Shift clustering algorithm with an Epanechnikov Kernel for event store data.

As event data has a non-smooth probability density in x and y space, and the Mean Shift algorithm performs a gradient ascent, the quality of the detected clusters depends significantly on the selected bandwidth hyperparameter, as well as the underlying data and the selected kernel. Generally the Gaussian Kernel yields better results for this kind of data, however it comes with a bigger performance impact.

The Mean Shift algorithm is an nonparametric estimate of the modes of the underlying probability distribution for the data. It implements an iterative search, starting from points provided by the user, or randomly selected from the data points provided. For each iteration, the current estimate of the mode is replaced by an estimate of the mean value of the surrounding data samples. If the Epanechnikov kernel is used for the underlying density estimate, its so-called "shadow kernel", the flat kernel must be used for the estimate of the mean. This means, that we can simply compute the average value of the data points that lie within a given radius around the current estimate of the mode, and use this as the next estimate. To provide an efficient search for the neighbours of the current mode estimate, a KD tree was used.

For the underlying theory, see "The Estimation of the Gradient of a Density Function with

Applications in Pattern Recognition" by K. Fukunaga and L. Hostetler as well as "Mean shift, mode seeking, and

clustering" by Yizong Cheng.

Public Types

using **Vector** = *dv*::*TimedKeyPoint*

using **VectorOfVectors** = *std*::vector<*Vector*, Eigen::aligned_allocator<*Vector*>>

Public Functions

inline **MeanShiftEventStoreAdaptor**(const *dv*::*EventStore* &data, const int16_t bw, float conv, const uint32_t maxIter, const *VectorOfVectors* &startingPoints, const uint32_t numLeaves = 32768)

Constructor

See also:

dv::containers::KDTree

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **bw** – The bandwidth used for the shift. This is a hyperparameter for the kernel. For the Epanechnikov kernel this means that all values within a radius of bw are averaged.
- **conv** – For each starting point, the algorithm is stopped as soon as the absolute value of the shift is <= conv.
- **maxIter** – The maximum number of iterations. Detected modes, for which the the number of iterations exceed this value are not added to the detected clusters.
- **startingPoints** – Points from which to start the search.
- **numLeaves** – the maximum number of leaves for the KDTree.

inline **MeanShiftEventStoreAdaptor**(const *dv*::*EventStore* &data, const int16_t bw, float conv, const uint32_t maxIter, *VectorOfVectors* &&startingPoints, const uint32_t numLeaves = 32768)

Constructor

See also:

dv::containers::KDTree

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **bw** – The bandwidth used for the shift. This is a hyperparameter for the kernel. For the Epanechnikov kernel this means that all values within a radius of bw are averaged.
- **conv** – For each starting point, the algorithm is stopped as soon as the absolute value of the shift is <= conv.
- **maxIter** – The maximum number of iterations. Detected modes, for which the the number of iterations exceed this value are not added to the detected clusters.
- **startingPoints** – Points from which to start the search.
- **numLeaves** – the maximum number of leaves for the KDTree.

inline **MeanShiftEventStoreAdaptor**(const *dv*::*EventStore* &data, const int16_t bw, float conv, const uint32_t maxIter, const uint32_t numStartingPoints, const uint32_t numLeaves = 32768)

Constructor

See also:

dv::containers::KDTree

Parameters

- **data** – The Matrix containing the data. The data is neither copied nor otherwise managed, ownership remains with the user of this class.
- **bw** – The bandwidth used for the shift. This is a hyperparameter for the kernel. For the Epanechnikov kernel this means that all values within a radius of bw are averaged.
- **conv** – For each starting point, the algorithm is stopped as soon as the absolute value of the shift is <= conv.

- **maxIter** – The maximum number of iterations. Detected modes, for which the the number of iterations exceed this value are not added to the detected clusters.
- **numStartingPoints** – The number of points which are randomly selected from the data points, to be used as starting points.
- **numLeaves** – the maximum number of leaves for the KDTree.

MeanShiftEventStoreAdaptor() = delete

MeanShiftEventStoreAdaptor(const *MeanShiftEventStoreAdaptor* &other) = delete

MeanShiftEventStoreAdaptor(*MeanShiftEventStoreAdaptor* &&other) = delete

MeanShiftEventStoreAdaptor &**operator=**(const *MeanShiftEventStoreAdaptor* &other) = delete

MeanShiftEventStoreAdaptor &**operator=**(*MeanShiftEventStoreAdaptor* &&other) = delete

~MeanShiftEventStoreAdaptor() = default

template<*kernel*::MeanShiftKernel **kernel** = *kernel*::Epanechnikov>
inline auto **fit**()

Executes the algorithm.

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Returns
The centres of each detected cluster

Returns
The labels for each data point. The labels correspond to the index of the centre to which the sample is assigned.

Returns
The number of samples in each cluster

Returns
The in-cluster variance for each cluster

template<*kernel*::MeanShiftKernel **kernel**>
inline *VectorOfVectors* **findClusterCentres**()

Performs the search for the cluster centres for each given starting point. A detected centre is added to the set of centres if it isn't closer than the bandwidth to any previously detected centre.

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Returns
The centres of each detected cluster

inline *std*::tuple<*std*::vector<uint32_t>, *std*::vector<uint32_t>, *std*::vector<float>> **assignClusters**(const *VectorOfVectors* &clusterCentres)

Assigns the data samples to a cluster by means of a nearest neighbour search, and computes the number of samples as well as the in-cluster variance in the process.

Parameters
clusterCentres – The centres of each detected cluster

Returns
The labels for each data point. The labels correspond to the index of the centre to which the sample is assigned.

Returns
The number of samples in each cluster

Returns
The in-cluster variance for each cluster

Public Static Functions

static inline *VectorOfVectors* **generateStartingPointsFromData**(const uint32_t numStartingPoints, const *dv*::*EventStore* &data)

Generates a vector of vectors containing the starting points by randomly selecting from provided data

Parameters

- **numStartingPoints** – The number of points to be generated
- **data** – the matrix to select the starting points from

Returns
The vector of vectors containing the starting points.

static inline *VectorOfVectors* **generateStartingPointsFromRange**(const uint32_t numStartingPoints, const *std*::array<*std*::pair<int16_t, int16_t>, 2> &ranges)

Generates a vector of vectors containing the starting points by generating random points within a given range for each dimension

Parameters

- **numStartingPoints** – The number of points to be generated
- **ranges** – a vector containing one range per dimension. Each dimension is represented by a pair containing the beginning and the end of the range

Returns
The vector of vectors containing the starting points.

Private Types

using **KDTree** = *dv*::*containers*::*kd_tree*::*KDTreeEventStoreAdaptor*

Private Functions

template<*kernel*::MeanShiftKernel **kernel**>
inline *std*::optional<*Vector*> **performShift**(*Vector* currentMode)

Performs a search for a mode in the underlying density starting off with a provided initial point.

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Parameters
currentMode – The starting point that is to be shifted until convergence.

Returns
An std::optional containing either a vector, if the search has converged, std::nullopt otherwise

template<*kernel*::MeanShiftKernel **kernel**>
inline float **applyKernel**(const float squaredDistance) const

Applies the selected kernel to the squared distance

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Parameters
squaredDistance – the squared distance between the current mode estimate and a given sample point

Returns
the kernel value

template<*kernel*::MeanShiftKernel **kernel**>
inline auto **getNeighbours**(const *Vector* &centre)

Returns the neighbours surrounding a centre

See also:

MeanShiftKernel

Template Parameters
kernel – the kernel to be used.

Parameters
centre – the centre surrounding which the neighbours are to be found

Returns
the neighbours, as a vector of pairs, one pair per neighbour containing a pointer to the event and a distance to the centre

inline float **squaredDistance**(const *dv*::*TimedKeyPoint* &k, const *dv*::Event &e) const

inline float **squaredDistance**(const *dv*::*TimedKeyPoint* &k1, const *dv*::*TimedKeyPoint* &k2) const

inline float **squaredDistance**(const *dv*::Event &e1, const *dv*::Event &e2) const

template<typename **T**>
inline *T* **pow2**(const *T* val) const

Private Members

const size_t **mNumSamples**

KDTree **mData**

const int16_t **mBandwidth**

const uint32_t **mMaxIter**

const float **mConvergence**

const *VectorOfVectors* **mStartingPoints**

Private Static Functions

static inline *Vector* **getZeroVector**()

class **MeanShiftTracker** : public *dv*::*features*::*TrackerBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/mean_shift_tracker.hpp> Track event blobs using mean shift algorithm on time surface event data.

Public Functions

inline **MeanShiftTracker**(const cv::Size &resolution, const int bandwidth, const *dv*::*Duration* timeWindow, *RedetectionStrategy*::*UniquePtr* redetectionStrategy = nullptr, *std*::unique_ptr<*EventFeatureBlobDetector*> detector = nullptr, const float stepSize = 0.5f, const float weightMultiplier = 1.f, float convergenceNorm = 0.01f, int maxIters = 2000)

Constructor for mean shift tracker using Epanechnikov kernel as weights for the time surface of events used to update track location. The kernel weights have highest value on the previous track location. This assumption is based on the idea that the new track location is "close" to last track location. The consecutive track updates are performed until the maximum number of iteration is reached or the shift between consecutive updates is below a threshold.

Parameters

- **resolution** – full image plane resolution

- **bandwidth** – search window dimension size. The search area is a square. The square side is 2 * bandwidth and center the current track location
- **timeWindow** – look back time from latest event: used to generate normalized time surface. All events older than (latestEventTime-timeWindow) will be discarded
- **redetectionStrategy** – strategy used to decide if and when to re-detect interesting points to track
- **detector** – detector used to re-detect tracks if redetection strategy uis defined and should happen
- **stepSize** – weight applied to shift to compute new track location. This value is in range (0, 1). A value of 0 means that no shift is performed. A value of 1 means that the new candidate center is directly assigned as new center
- **weightMultiplier** – scaling factor for Epanechnikov weights used in the computation of the mean shift cost update
- **convergenceNorm** – shift value below which search will not continue (this value is named "mode" in the docs)
- **maxIters** – maximum number of search iterations for one track update

inline void **accept**(const *dv*::*EventStore* &store)

Add events to time surface and update last batch of events fed to the tracker.

Parameters
store – new incoming events for the tracker.

inline virtual Result::SharedPtr **track**() override

Compute new centers based on area with highest event density. The density is weighted by the event timestamp: newer timestamps have higher weight.

Returns
structure containing new track locations as a vector of *dv::TimedKeyPoint*

inline void **setRedetectionStrategy**(*RedetectionStrategy*::*UniquePtr* redetectionStrategy)

Define redetection strategy used to re-detect interesting points to track.

Parameters
redetectionStrategy – type of redetection to use (check redetection_strategy.hpp for available types of re-detections)

inline void **setDetector**(*std*::unique_ptr<*EventFeatureBlobDetector*> detector)

Define detector used to detect interesting points to track (if redetection should happen)

Parameters
detector – detector for new interesting points to track

inline int **getBandwidth**() const

Getter for bandwidth value that defines the search area for a new track. For detailed information on how the area is computed please check related parameter in constructor.

Returns
search window dimension size.

inline void **setBandwidth**(const int bandwidth)

Setter for bandwidth value.

Parameters
bandwidth – search window dimension size.

inline *dv*::*Duration* **getTimeWindow**() const

Get time window duration used to normalize time surface.

Returns
value of time window use to generate normalized time surface

inline void **setTimeWindow**(const *dv*::*Duration* timeWindow)

Setter for time window duration for time surface normalization.

Parameters
timeWindow – size of window

inline float **getStepSize**() const

Get multiplier value used for track location update. Given a computed shift to be applied to a track, the actual shift performed is given by mStepSize * shift.

Returns
scaling value applied to the spatial interval computed between current and new track position at consecutive updates

inline void **setStepSize**(const float stepSize)

Setter for learning rate for motion towards new center during one mean shift iteration. Please check the same parameter in the constructor description for detailed information.

Parameters
stepSize – weight applied to shift to compute new track location.

inline float **getWeightMultiplier**() const

Getter for weight multiplier used to adjust weight of each time surface value in the mean shift update. If multiplier is smaller than 1, the cost values for each location are shrink, whereas if the multiplier is larger than 1, the difference between time surface intensities will be larger.

Returns
weight multiplier value

inline void **setWeightMultiplier**(const float multiplier)

Setter for scaling factor used in the computation of the mean shift cost update.

Parameters
multiplier – scaling factor value

inline float **getConvergenceNorm**() const

Get norm of distance between consecutive tracks updates. If the distance is smaller than this norm, the track update is considered to be converged.

Returns
value of distance norm between consecutive updates

inline void **setConvergenceNorm**(const float norm)

Setter for threshold norm (i.e. mode) between consecutive track updates below which iterations are stopped.

Parameters
norm – threshold value

inline int **getMaxIterations**() const

Get maximum number of times track update can be run.

Returns
value of maximum number of operations for track update

inline void **setMaxIterations**(const int maxIters)

Setter for maximum number of track updates.

Parameters

maxIters – value of maximum number of operations for track update

Private Functions

inline Result::SharedPtr **updateTracks**(const cv::Mat &normalizedTimeSurface)

Compute new location for all tracks. If a new position fall inside the area of a new position computed for a previous track, the track will not be updated. Previous track with its timestamp will be kept.

Parameters

normalizedTimeSurface – image representation of event timestamps based on time surface

Returns

updated track positions

inline *std*::optional<*dv*::Point2f> **computeShift**(const *dv*::Point2f &center, const cv::Mat &timeSurface, const float trackSize)

Compute new track location. Note: kernel weights are updated only if the search window changed size or if it intersects the boundaries of the image plane. This decision has been made for performance reasons and should not affect the final result as long as the new track position is "close enough", to the starting position.

Parameters

- **center** – previous track location
- **timeSurface** – Matrix containing normalized time surface values
- **trackSize** – dimension of track determining kernel size

Returns

new final track location if value is valid, std::nullopt is returned if the search area has no event data inside it.

inline *std*::optional<*dv*::Point2f> **updateCenterLocation**(const cv::Mat &spatialWindow, const cv::Mat &kernelWeights) const

Compute mode (i.e. track location).

Parameters

- **spatialWindow** – image plane sub-matrix in which the center will be updated
- **kernelWeights** – weights of Epanechnikov kernel applied to each time surface location inside the given spatial window

Returns

new track location

inline cv::Mat **kernelEpanechnikovWeights**(const *dv*::Point2f &center, const cv::Rect &window, const float cutOffValue) const

Compute Epanechnikov kernel with highest peak at center location.

Parameters

- **center** – location of the observed sample, i.e. location where kernel will have highest response
- **window** – window in which kernel weights will be computed

- **cutOffValue** – distance from center after which all kernel weights will be zero. For more information about the kernel please check equation 3.54 from http://sfb649.wiwi.hu-berlin.de/fedc_homepage/xplore/ebooks/html/spm/spmhtmlnode18.html

Returns
matrix with weights of Epanechnikov kernel

inline *std*::pair<cv::Mat, cv::Rect> **findSpatialWindow**(const *dv*::Point2f &center, const cv::Mat &image) const

Compute area in which the new track position will be searched. This area depends on the bandwidth value. The search area is defined as the square around the center value with size of one side as 2*bandwidth. We return the selected area as first argument and the roi in the full image plane to be able to retrieve coordinates of selected area in the original image space.

Parameters

- **center** – previous track center around which we define the search area
- **image** – full image plane data

Returns
pair containing as first output the matrix block containing the data inside the image defined by the rectangle returned as second output

inline void **runRedetection**(Result::SharedPtr &result)

Re-detect interesting points

Parameters
result – current set of tracks to which new detections will be added

Private Members

int **mBandwidth**

parameter defining search window size for each track update

dv::*TimeSurface* **mSurface**

event time surface

dv::*Duration* **mTimeWindow**

time window of events to generate the normalized time surface from

float **mStepSize**

cv::Size **mResolution**

dv::*EventStore* **mEvents** = *dv*::*EventStore*()

latest batch of events fed to the tracker

std::unique_ptr<*EventFeatureBlobDetector*> **mDetector**

detector used if no track has been detected or redetection is expected to happen

int32_t **mLastFreeClassId** = 0

value used to keep track of first free ID for a new track

RedetectionStrategy::*UniquePtr* **mRedetectionStrategy** = nullptr

type of redetection strategy used to detect new interesting points to track

float **mWeightMultiplier**

Weight multiplier used to adjust weight of each point in the mean shift update. If multiplier is smaller than 1, the cost values for each location are shrink, whereas if the multiplier is larger than 1, the difference between points with lower intensity in the time surface will be increased from ones with larger intensity values

float **mConvergenceNorm**

shift value below which search will not continue

int **mMaxIters**

maximum number of search iterations for one track update

class **Metadata**

Public Functions

Metadata() = default

inline **Metadata**(const cv::Size &patternShape_, const cv::Size &internalPatternShape_, const *std*::string &patternType_, float patternSize_, float patternSpacing_, const *std*::optional<float> &calibrationError_, const *std*::string &calibrationTime_, const *std*::string &quality_, const *std*::string &comment_, const *std*::optional<float> &pixelPitch_)

inline explicit **Metadata**(const pt::ptree &tree)

Create an instance of metadata from a property tree structure.

Parameters
tree – Property tree to be parsed.

Returns
Constructed *Metadata* instance.

inline pt::ptree **toPropertyTree**() const

Serialize the metadata structure into a property tree.

Returns
Serialized property tree.

inline bool **operator==**(const *Metadata* &rhs) const

Equality operator.

Parameters
rhs –

Returns

Public Members

cv::Size **patternShape**

Shape of the calibration pattern.

cv::Size **internalPatternShape**

Shape of the calibration pattern in terms of internal intersections.

std::string **patternType**

Type of the calibration pattern used (e.g. apriltag)

float **patternSize** = -1.f

Size of the calibration pattern in [m].

float **patternSpacing** = -1.f

Ratio between tags to patternSize (apriltag only)

std::optional<float> **calibrationError** = *std*::nullopt

Calibration reprojection error.

std::string **calibrationTime**

Timestamp when the calibration was conducted.

std::string **quality**

Description of the calibration quality (excellent/good/bad etc)

std::string **comment**

Any additional information.

std::optional<float> **pixelPitch** = *std*::nullopt

Pixel pitch in meters.

struct **Metadata**

Public Functions

inline explicit **Metadata**(const *std*::string &calibrationTime = "", const *std*::string &comment = "")

inline explicit **Metadata**(const pt::ptree &tree)

inline pt::ptree **toPropertyTree**() const

inline bool **operator==**(const *Metadata* &rhs) const

Public Members

std::string **calibrationTime**

Timestamp when the calibration was conducted.

std::string **comment**

Any additional information.

struct **Metadata**

#include *</builds/inivation/dv/dv-processing/include/dv-processing/camera/calibrations/stereo_calibration.hpp>*
Metadata for the stereo calibration.

Public Functions

Metadata() = default

inline explicit **Metadata**(const *std*::optional<float> &epipolarError, const *std*::string_view comment = "")

inline explicit **Metadata**(const pt::ptree &tree)

inline pt::ptree **toPropertyTree**() const

Serialize into a property tree.

Returns

inline bool **operator==**(const *Metadata* &rhs) const

Public Members

std::optional<float> **epipolarError** = *std*::nullopt

Average epipolar error.

std::string **comment**

Any additional information.

class **MonoCameraRecording** : public *dv*::*io*::*CameraInputBase*

#include *</builds/inivation/dv/dv-processing/include/dv-processing/io/mono_camera_recording.hpp>* A convenience class for reading recordings containing data captured from a single camera. Looks for an event, frame, imu, and trigger streams within the supplied aedat4 file.

Public Functions

inline explicit **MonoCameraRecording**(const *std*::shared_ptr<*ReadOnlyFile*> &fileReader, const *std*::string &cameraName = "")

Create a reader that reads single camera data recording from a pre-constructed file reader.

Parameters

- **fileReader** – A pointer for pre-constructed file reader.
- **cameraName** – Name of the camera in the recording. If an empty string is passed (the default value), reader will try detect the name of the camera. In case recording contains more than one camera, it will choose the first encountered name and ignore streams that were recorded by a different camera.

inline explicit **MonoCameraRecording**(const fs::path &aedat4Path, const *std*::string &cameraName = "")

Create a reader that reads single camera data recording from an aedat4 file.

Parameters

- **aedat4Path** – Path to the aedat4 file.
- **cameraName** – Name of the camera in the recording. If an empty string is passed (the default value), reader will try detect the name of the camera. In case recording contains more than one camera, it will choose the first encountered name and ignore streams that were recorded by a different camera.

inline virtual *std*::optional<*dv*::*Frame*> **getNextFrame**() override

Sequential read of a frame, tries reading from stream named "frames". This function increments an internal seek counter which will return the next frame at each call.

Returns
A *dv::Frame* or `std::nullopt` if the frame stream is not available or the end-of-stream was reached.

inline *std*::optional<*dv*::*Frame*> **getNextFrame**(const *std*::string &streamName)

Sequential read of a frame. This function increments an internal seek counter which will return the next frame at each call.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with frame data type.

Returns
A *dv::Frame*, `std::nullopt` if the frame stream is not available or the end-of-stream was reached.

inline bool **isStreamAvailable**(const *std*::string &streamName)

Check whether a given stream name is available.

Parameters
streamName – Name of the stream.

Returns
True if this stream is available, false otherwise.

inline *std*::vector<*std*::string> **getStreamNames**() const

Return a vector containing all available stream names.

Returns
A list of custom data type stream names.

template<class **DataType**>
inline *std*::optional<*DataType*> **getNextStreamPacket**(const *std*::string &streamName)

Read a custom data type packet sequentially.

Custom data types are any flatbuffer generated types that are not the following: *dv::EventPacket*, *dv::TriggerPacket*, *dv::IMUPacket*, *dv::Frame*.

Template Parameters
DataType – Custom data packet class.

Parameters
streamName – Name of the stream.

Throws

- `InvalidArgument` – An exception is thrown if a stream with given name is not found in the file.
- `InvalidArgument` – An exception is thrown if given type does not match the type identifier of the given stream.

Returns
Next packet within given stream or `std::nullopt` in case of end-of-stream.

inline virtual *std*::optional<*dv*::*EventStore*> **getNextEventBatch**() override

Sequential read of events, tries reading from stream named "events". This function increments an internal seek counter which will return the next event batch at each call.

Returns
A `dv::EventStore` or `std::nullopt` if the frame stream is not available or the end-of-stream was reached.

inline *std*::optional<*dv*::*EventStore*> **getNextEventBatch**(const *std*::string &streamName)

Sequentially read a batch of recorded events. This function increments an internal seek counter which will return the next batch at each call.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with event data type.

Returns
A vector containing events, `std::nullopt` if the event stream is not available or the end-of-stream was reached.

inline virtual *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **getNextImuBatch**() override

Sequential read of imu data, tries reading from stream named "imu". This function increments an internal seek counter which will return the next imu data batch at each call.

Returns
A vector or *IMU* measurements or `std::nullopt` if the imu data stream is not available or the end-of-stream was reached.

inline *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **getNextImuBatch**(const *std*::string &streamName)

Sequentially read a batch of recorded imu data. This function increments an internal seek counter which will return the next batch at each call.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with imu data type.

Returns

A vector containing imu data, `std::nullopt` if the imu data stream is not available or the end-of-stream was reached.

inline virtual *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **getNextTriggerBatch**() override

Sequential read of trigger data, tries reading from stream names "triggers". This function increments an internal seek counter which will return the next trigger data batch at each call.

Returns

A vector of trigger data or `std::nullopt` if the frame stream is not available or the end-of-stream was reached.

inline *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **getNextTriggerBatch**(const *std*::string &streamName)

Sequentially read a batch of recorded triggers. This function increments an internal seek counter which will return the next batch at each call.

Parameters

streamName – Name of the stream, if an empty name is passed, it will select any one stream with trigger data type.

Returns

A vector containing triggers, `std::nullopt` if the trigger stream is not available or the end-of-stream was reached.

inline void **resetSequentialRead**()

Reset the sequential read function to start from the beginning of the file.

inline virtual bool **isRunning**() const override

Check whether sequential read functions has not yet reached end-of-stream.

Returns

True if at least one of the streams has reached end-of-stream, false otherwise.

inline *std*::optional<*dv*::*EventStore*> **getEventsTimeRange**(const int64_t startTime, const int64_t endTime, const *std*::string &streamName = "events")

Get events within given time range [startTime; endTime).

Parameters

- **startTime** – Start timestamp of the time range.
- **endTime** – End timestamp of the time range.
- **streamName** – Name of the stream, if an empty name is passed, it will select any one stream with event data type.

Returns

`dv::EventStore` with events in the time range if the event stream is available, `std::nullopt` otherwise.

inline *std*::optional<*dv*::*cvector*<*dv*::*Frame*>> **getFramesTimeRange**(const int64_t startTime, const int64_t endTime, const *std*::string &streamName = "frames")

Get frames within given time range [startTime; endTime).

Parameters

- **startTime** – Start timestamp of the time range.
- **endTime** – End timestamp of the time range.

- **streamName** – Name of the stream, if an empty name is passed, it will select any one stream with frame data type.

Throws

`InvalidArgument` – If frame stream doesn't exists or a stream with given name doesn't exist.

Returns

Vector containing frames and timestamps.

template<class **DataType**>
inline *std*::optional<*dv*::*cvector*<*DataType*>> **getStreamTimeRange**(const int64_t startTime, const int64_t endTime, const *std*::string &streamName)

Get packets from a stream within given period of time. Returns a vector of packets. If a packet contains elements that are outside of given time range, the internal elements will be cut to match exactly the [startTime; endTime). If stream does not contain any packets within requested time range, the function returns an empty vector.

Template Parameters

DataType – Packet type

Parameters

- **startTime** – Period start timestamp.
- **endTime** – Period end timestamp.
- **streamName** – Name of the stream, empty string will pick a first stream with matching type.

Throws

- `InvalidArgument` – An exception is thrown if a stream with given name is not found in the file.
- `InvalidArgument` – An exception is thrown if given type does not match the type identifier of the given stream.

Returns

A vector of packets containing the data only within [startTime; endTime) period.

inline *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **getImuTimeRange**(const int64_t startTime, const int64_t endTime, const *std*::string &streamName = "imu")

Get *IMU* data within given time range [startTime; endTime).

Parameters

- **startTime** – Start timestamp of the time range.
- **endTime** – End timestamp of the time range.
- **streamName** – Name of the stream, if an empty name is passed, it will select any one stream with imu data type.

Returns

Vector containing *IMU* data if the *IMU* stream is available, `std::nullopt` otherwise.

inline *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **getTriggersTimeRange**(const int64_t startTime, const int64_t endTime, const *std*::string &streamName = "triggers")

Get trigger data within given time range [startTime; endTime).

Parameters

- **startTime** – Start timestamp of the time range.
- **endTime** – End timestamp of the time range.
- **streamName** – Name of the stream, if an empty name is passed, it will select any one stream with trigger data type.

Returns
Vector containing triggers if the trigger stream is available, `std::nullopt` otherwise.

inline virtual bool **isFrameStreamAvailable**() const override

Check whether frame stream is available. Specifically checks whether a stream named "frames" is available since it's the default stream name for frames.

Returns
True if the frame stream is available.

inline bool **isFrameStreamAvailable**(const *std*::string &streamName) const

Checks whether a frame data stream is present in the file.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with frame data type.

Returns
True if the frames are available, false otherwise.

inline virtual bool **isEventStreamAvailable**() const override

Check whether event stream is available. Specifically checks whether a stream named "events" is available since it's the default stream name for events.

Returns
True if the event stream is available, false otherwise.

inline bool **isEventStreamAvailable**(const *std*::string &streamName) const

Checks whether an event data stream is present in the file.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with event data type.

Returns
True if the events are available, false otherwise.

inline virtual bool **isImuStreamAvailable**() const override

Check whether imu data stream is available. Specifically checks whether a stream named "imu" is available since it's the default stream name for imu data.

Returns
True if the imu stream is available, false otherwise.

inline bool **isImuStreamAvailable**(const *std*::string &streamName) const

Checks whether an imu data stream is present in the file.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with *IMU* data type.

Returns
True if the imu data is available, false otherwise.

inline virtual bool **isTriggerStreamAvailable**() const override

Check whether trigger stream is available. Specifically checks whether a stream named "triggers" is available since it's the default stream name for trigger data.

Returns
True if the trigger stream are available, false otherwise.

inline bool **isTriggerStreamAvailable**(const *std*::string &streamName) const

Checks whether a trigger data stream is present in the file.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with trigger data type.

Returns
True if the triggers are available, false otherwise.

inline *std*::pair<int64_t, int64_t> **getTimeRange**() const

Return a pair containing start (first) and end (second) time of the recording file.

Returns
A pair containing start and end timestamps for the recording.

inline *dv*::*Duration* **getDuration**() const

Return the duration of the recording.

Returns
Duration value holding the total playback time of the recording.

inline virtual *std*::string **getCameraName**() const override

Return the camera name that is detected in the recording.

Returns
String containing camera name.

inline *DataReadVariant* **readNext**()

Read next packet in the recorded stream, the function returns a `std::variant` containing one of the following types:

- dv::EventStore
- *dv::Frame*
- dv::cvector<dv::IMU>
- dv::cvector<dv::Trigger>
- dv::io::MonoCameraRecording::OutputFlag The `OutputFlag` is used to determine when the end of file is reached. If the reader encounters an unsupported type, the data will be skipped and will seek until a packet containing a supported type is reached.

Returns
`std::variant` containing a packet with data of one of the supported types.

inline bool **handleNext**(*DataReadHandler* &handler)

Read next packet from the recording and use a handler object to handle all types of packets. The function returns a true if end-of-file was not reached, so this function call can be used in a while loop like so:

```
while (recording.handleNext(handler)) {
        // While-loop executes after each packet
}
```

Parameters
handler –

Returns

inline void **run**(*DataReadHandler* &handler)

Sequentially read all packets from the recording and apply handler to each packet. This is a blocking call.

Parameters
handler – Handler class containing lambda functions for each supported packet type.

inline virtual *std*::optional<cv::Size> **getEventResolution**() const override

Get event stream resolution for the "events" stream.

Returns
Resolution of the "events" stream.

inline *std*::optional<cv::Size> **getEventResolution**(const *std*::string &streamName) const

Get the resolution of the event data stream if it is available.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with event data type.

Returns
Returns the resolution of the event data if available, `std::nullopt` otherwise.

inline virtual *std*::optional<cv::Size> **getFrameResolution**() const override

Get frame stream resolution for the "frames" stream.

Returns
Resolution of the "frames" stream.

inline *std*::optional<cv::Size> **getFrameResolution**(const *std*::string &streamName) const

Get the resolution of the frame data stream if it is available.

Parameters
streamName – Name of the stream, if an empty name is passed, it will select any one stream with frame data type.

Returns
Returns the resolution of the frames if available, `std::nullopt` otherwise.

inline const *std*::map<*std*::string, *std*::string> &**getStreamMetadata**(const *std*::string &streamName)

Get all metadata of a stream.

Parameters
streamName – Name of the stream.

Throws
`out_of_range` – Out of range exception is thrown if a stream with given name is not available.

Returns
A map containing key-value strings of each available metadata of a requested stream.

inline *std*::optional<*std*::string> **getStreamMetadataValue**(const *std*::string &streamName, const *std*::string &key)

Get a value of a given metadata key. Throws an exception if given stream doesn't exist and returns std::nullopt if a metadata entry with given key is not found for the stream.

Parameters

- **streamName** – Name of the stream.
- **key** – Key string of the metadata.

Throws

`out_of_range` – Out of range exception is thrown if a stream with given name is not available.

Returns

Metadata entry with given key is found for the stream, `std::nullopt` otherwise.

template<class **DataType**>
inline bool **isStreamOfDataType**(const *std*::string &streamName) const

Check whether a stream is of a given data type.

Template Parameters

DataType – Data type to be checked.

Parameters

streamName – Name of the stream.

Throws

`out_of_bounds` – Out of bounds exception is thrown if stream of a given name is not found.

Returns

True if the given stream contains DataType data.

Private Types

typedef *std*::map<*std*::string, *StreamDescriptor*> **StreamInfoMap**

Private Functions

inline const *dv*::*io*::*Stream* ***getStream**(const int streamId) const

inline void **parseStreamIds**()

template<class **DataType**>
inline *StreamInfoMap*::iterator **getStreamInfo**(const *std*::string &streamName)

template<class **DataType**>
inline *StreamInfoMap*::const_iterator **getStreamInfo**(const *std*::string &streamName) const

template<class **DataType**>
inline *std*::shared_ptr<*DataType*> **getNextPacket**(*StreamDescriptor* &streamInfo)

Private Members

std::shared_ptr<*ReadOnlyFile*> **mReader** = nullptr

FileInfo **mInfo**

std::string **mCameraName**

dv::*cvector*<*FileDataDefinition*>::const_iterator **mPacketIter**

bool **eofReached** = false

StreamInfoMap **mStreamInfo**

Private Static Functions

template<class **VectorClass**>
static inline void **trimVector**(*VectorClass* &vector, int64_t start, int64_t end)

Trim a vector containing elements with a timestamp. Retains only the data within [start; end).

Template Parameters
VectorClass – The class of the vector

Parameters

- **vector** – The vector of data
- **start** – Start timestamp (inclusive start of range)
- **end** – End timestamp (exclusive end of range)

class **MonoCameraWriter**

Public Functions

inline **MonoCameraWriter**(const fs::path &aedat4Path, const *MonoCameraWriter*::Config &config, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Create an aedat4 file writer with simplified API.

Parameters

- **aedat4Path** – Path to the output file. The file is going to be overwritten.
- **config** – *Writer* config. Defines expected output streams and recording metadata.
- **resolver** – Type resolver for the output file.

inline **MonoCameraWriter**(const fs::path &aedat4Path, const *CameraCapture* &capture, const *CompressionType* compression = *CompressionType*::*LZ4*, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Create an aedat4 file writer that inspects the capabilities and configuration from a `dv::io::CameraCapture` class. This will enable all available data streams present from the camera capture.

Parameters

- **aedat4Path** – Path to the output file. The file is going to be overwritten.
- **capture** – Direct camera capture instance. This is used to inspect the available data streams and metadata of the camera.
- **compression** – Compression to be used for the output file.
- **resolver** – Type resolver for the output file.

inline void **writeEventPacket**(const *dv*::*EventPacket* &events, const *std*::string &streamName = "events")

Write an event packet into the output file.

The data is passed directly into the serialization procedure without performing copies. Data is serialized and the actual file IO is performed on a separate thread.

Parameters

- **events** – Packet of events.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws

`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was not added during construction.

inline void **writeEvents**(const *dv*::*EventStore* &events, const *std*::string &streamName = "events")

Write an event store into the output file. The store is written by maintaining internal data partial ordering and fragmentation.

The data is passed directly into the serialization procedure without performing copies. Data is serialized and the actual file IO is performed on a separate thread.

Parameters

- **events** – Store of events.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws

`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was not added during construction.

inline void **writeFrame**(const *dv*::*Frame* &frame, const *std*::string &streamName = "frames")

Write a frame image into the file.

The data is passed directly into the serialization procedure without performing copies. Data is serialized and the actual file IO is performed on a separate thread.

NOTE: if the frame contains an empty image, it will be ignored and not recorded.

Parameters

- **frame** – A frame to be written.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws

`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was not added during construction.

inline void **writeImuPacket**(const *dv*::*IMUPacket* &packet, const *std*::string &streamName = "imu")

Write a packet of imu data into the file.

The data is passed directly into the serialization procedure without performing copies. Data is serialized and the actual file IO is performed on a separate thread.

Parameters

- **packet** – *IMU* measurement packet.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws

`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was not added during construction.

inline void **writeImu**(const *dv*::*IMU* &imu, const *std*::string &streamName = "imu")

Write an *IMU* measurement.

This function is not immediate, it batches the measurements until a configured amount is reached, only then the data is passed to the serialization step. Only then the data will be passed to the file write IO thread. If the file is closed (the object gets destroyed), destructor will dump the rest of the buffered measurements to the serialization step.

See also:

setPackagingCount

Parameters

- **imu** – A single *IMU* measurement.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws

`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was added enabled during construction.

inline void **writeTriggerPacket**(const *dv*::*TriggerPacket* &packet, const *std*::string &streamName = "triggers")

Write a packet of trigger data into the file.

The data is passed directly into the serialization procedure without performing copies. Data is serialized and the actual file IO is performed on a separate thread.

Parameters

- **packet** – *Trigger* data packet.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws

`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was added enabled during construction.

template<class **PacketType**>
inline void **writePacket**(const *PacketType* &packet, const *std*::string &stream)

Write a packet into a named stream.

Template Parameters
PacketType – Type of data packet.

Parameters

- **stream** – Name of the stream, an empty string will match first stream with compatible data type.
- **packet** – Data packet

Throws

- `InvalidArgument` – If a stream with given name is not configured.
- `InvalidArgument` – If a stream with given name is configured for a different type of data packet.
- `invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was added enabled during construction.

inline void **writeTrigger**(const *dv*::*Trigger* &trigger, const *std*::string &streamName = "triggers")

Write a *Trigger* measurement.

This function is not immediate, it batches the measurements until a configured amount is reached, only then the data is passed to the serialization step. Only then the data will be passed to the file write IO thread. If the file is closed (the object gets destroyed), destructor will dump the rest of the buffered measurements to the serialization step.

See also:

setPackagingCount

Parameters

- **imu** – A single *Trigger* measurement.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

Throws
`invalid_argument` – Invalid argument exception is thrown if function is called and compatible output stream was not added during construction.

template<class **PacketType**, class **ElementType**>
inline void **writePacketElement**(const *ElementType* &element, const *std*::string &streamName)

Write a single element into packet. A packet will be created per stream and element will be added until packaging count is reached, at that point the packet will be written do disk.

Template Parameters

- **PacketType** – Type of the packet to hold the elements.
- **ElementType** – Type of an element.

Parameters

- **element** – Element to be saved.
- **streamName** – Name of the stream, an empty string will match first stream with compatible data type.

inline void **setPackagingCount**(size_t packagingCount)

Set the size batch size for trigger and imu buffering. The single measurements passed into `writeTrigger` and `writeImu` functions will packed into batches of the given size before writing to the file.

A packaging value of 0 or 1 will cause each measurement to be serialized immediately.

See also:

writeTrigger

See also:

writeImu

Parameters
packagingCount – *Trigger* and *IMU* measurement packet size that is batched up using the `writeImu` and `writeTrigger` functions.

inline bool **isEventStreamConfigured**(const *std*::string &streamName = "events") const

Check if the event stream is configured for this writer.

Parameters
streamName – Name of the stream, an empty string will match first stream with compatible data type.

Returns
True if event stream is configured, false otherwise.

inline bool **isFrameStreamConfigured**(const *std*::string &streamName = "frames") const

Check if the frame stream is configured for this writer.

Parameters
streamName – Name of the stream, an empty string will match first stream with compatible data type.

Returns
True if frame stream is configured, false otherwise.

inline bool **isImuStreamConfigured**(const *std*::string &streamName = "imu") const

Check if the *IMU* stream is configured for this writer.

Parameters
streamName – Name of the stream, an empty string will match first stream with compatible data type.

Returns
True if *IMU* stream is configured, false otherwise.

inline bool **isTriggerStreamConfigured**(const *std*::string &streamName = "triggers") const

Check if the trigger stream is configured for this writer.

Parameters
streamName – Name of the stream, an empty string will match first stream with compatible data type.

Returns
True if trigger stream is configured, false otherwise.

template<class **PacketType**>

inline bool **isStreamConfigured**(const *std*::string &streamName) const

Check whether a stream with given name and compatible data type is configured.

Template Parameters

PacketType – Type of the packet to hold the elements.

Parameters

streamName – Name of the stream, an empty string will match first stream with compatible data type.

Returns

inline **~MonoCameraWriter**()

Public Static Functions

static inline *Config* **EventOnlyConfig**(const *std*::string &cameraName, const cv::Size &resolution, *dv*::*CompressionType* compression = *dv*::*CompressionType*::*LZ4*)

Generate a config for a writer that will expect a stream of events only.

Parameters

- **cameraName** – Name of the camera.
- **resolution** – Camera sensor resolution.
- **compression** – Compression type.

Returns

A config template for *MonoCameraWriter*.

static inline *Config* **FrameOnlyConfig**(const *std*::string &cameraName, const cv::Size &resolution, *dv*::*CompressionType* compression = *dv*::*CompressionType*::*LZ4*)

Generate a config for a writer that will expect a stream of frames only.

Parameters

- **cameraName** – Name of the camera.
- **resolution** – Camera sensor resolution.
- **compression** – Compression type.

Returns

A config template for *MonoCameraWriter*.

static inline *Config* **DVSConfig**(const *std*::string &cameraName, const cv::Size &resolution, *dv*::*CompressionType* compression = *dv*::*CompressionType*::*LZ4*)

Generate a config for a writer that will expect data from a DVS camera - events, *IMU*, triggers.

Parameters

- **cameraName** – Name of the camera.
- **resolution** – Camera sensor resolution.
- **compression** – Compression type.

Returns

A config template for *MonoCameraWriter*.

static inline *Config* **DAVISConfig**(const *std*::string &cameraName, const cv::Size &resolution, *dv*::*CompressionType* compression = *dv*::*CompressionType*::*LZ4*)

Generate a config for a writer that will expect data from a DAVIS camera - frames, events, *IMU*, triggers.

Parameters

- **cameraName** – Name of the camera.
- **resolution** – Camera sensor resolution.
- **compression** – Compression type.

Returns
A config template for *MonoCameraWriter*.

static inline *Config* **CaptureConfig**(const *dv*::*io*::*CameraCapture* &capture, *dv*::*CompressionType* compression = *dv*::*CompressionType*::*LZ4*)

Generate a config from a camera capture instance, this only checks whether camera provides frame data stream or not and enables all available streams to be recorded.

Parameters

- **capture** – Camera capture class instance.
- **compression** – Compression type.

Returns
A config template for *MonoCameraWriter*.

Private Types

typedef *std*::map<*std*::string, *StreamDescriptor*> **StreamDescriptorMap**

Private Functions

inline *std*::string **createHeader**(const *MonoCameraWriter*::*Config* &config, const *dv*::*io*::*support*::*TypeResolver* &resolver)

template<class **PacketType**>
inline *StreamDescriptorMap*::iterator **findStreamDescriptor**(const *std*::string &streamName)

template<class **PacketType**>
inline *StreamDescriptorMap*::const_iterator **findStreamDescriptor**(const *std*::string &streamName) const

inline explicit **MonoCameraWriter**(const *std*::shared_ptr<*dv*::*io*::*WriteOnlyFile*> &outputFile, const *dv*::*io*::*MonoCameraWriter*::*Config* &config, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Preconfigured output file constructor. Internal use only, used for multi-camera recording.

Parameters

- **outputFile** – *WriteOnlyFile* instance to write data.
- **config** – Output stream configuration.
- **resolver** – Type resolver for the output file.

Private Members

size_t **mPackagingCount** = 20

MonoCameraWriter::*Config* **inputConfig**

StreamDescriptorMap **mOutputStreamDescriptors**

dv::*io*::*support*::*XMLTreeNode* **mRoot**

std::shared_ptr<*dv*::*io*::*WriteOnlyFile*> **mOutput**

Private Static Functions

static inline void **validateConfig**(const *MonoCameraWriter*::*Config* &config)

Friends

friend class StereoCameraWriter

template<class **Accumulator** = *dv*::*EdgeMapAccumulator*, class **PixelPredictor** = *kinematics*::*PixelMotionPredictor*>
class **MotionCompensator**

Public Functions

inline const Info &**getInfo**() const

Return an info class instance containing motion compensator state for the algorithm iteration. The info object contains debug information about the execution of the motion compensator.

Returns

inline void **accept**(const *Transformationf* &transform)

Push camera pose measurement.

Parameters

transform – Transform representing camera pose in some fixed reference frame (e.g. World coordinates).

inline void **accept**(const *dv*::*measurements*::*Depth* &timeDepth)

Scene depth measurement in meters.

Parameters

timeDepth – A pair containing measured depth into the scene and a timestamp at when the measurement was performed.

inline void **accept**(const *dv*::*EventStore* &events)

Push event camera input.

Parameters

events – Pixel brightness changes from an event camera.

inline void **accept**(const *dv*::Event &event)

Push event camera input.

Parameters
event – Pixel brightness change from an event camera.

inline *dv*::*EventStore* **generateEvents**(const int64_t generationTime = -1)

Generate the motion compensated events contained in the buffer.

Parameters
generationTime – Provide a timestamp to which point in time the motion compensator compensates into, negative values will cause the function to use highest timestamp value in the event buffer.

Returns
Motion compensated events.

inline *dv*::*Frame* **generateFrame**(const int64_t generationTime = -1)

Generate the motion compensated frame output and reset the events contained in the buffer.

Parameters
generationTime – Provide a timestamp to which point in time the motion compensator compensates into, negative values will cause the function to use highest timestamp value in the event buffer.

Returns
Motion compensated frame.

inline void **reset**()

Clear the event buffer.

inline *MotionCompensator* &**operator<<**(const *dv*::*EventStore* &store)

Accept the event data using the stream operator.

Parameters
store – Input event store.

Returns
Reference to current object instance.

inline *MotionCompensator* &**operator<<**(const *dv*::Event &event)

Accept the event data using the stream operator.

Parameters
store – Input event.

Returns
Reference to current object instance.

inline *dv*::*Frame* &**operator>>**(*dv*::*Frame* &image)

Output stream operator which generates a frame.

Parameters
image – Motion compensated frame.

Returns
Motion compensated frame.

inline **MotionCompensator**(const *camera*::*CameraGeometry*::*SharedPtr* &cameraGeometry,
std::unique_ptr<*Accumulator*> accumulator_)

Construct a motion compensator instance with custom accumulator.

Parameters

- **cameraGeometry** – Camera geometry class instance containing intrinsic calibration of the camera sensor.
- **accumulator_** – *Accumulator* instance to be used to accumulate events.

inline explicit **MotionCompensator**(const *camera*::*CameraGeometry*::*SharedPtr* &cameraGeometry)

Construct a motion compensator instance with default accumulator. Default accumulator is a `dv::EdgeMapAccumulator` with default parameters.

Parameters

cameraGeometry – Camera geometry class instance containing intrinsic calibration of the camera sensor.

inline explicit **MotionCompensator**(const cv::Size &sensorDimensions)

Construct a motion compensator with no known calibration. This assumes that the camera is an ideal pinhole camera sensor (no distortion) with focal length equal to camera sensor width in pixels and central point is the exact geometrical center of the pixel array.

Parameters

sensorDimensions – Camera sensor resolution.

inline float **getConstantDepth**() const

Get currently assumed constant depth value. It is used if no depth measurements are provided.

See also:

setConstantDepth

Returns

Currently used aistance to the scene (depth).

inline void **setConstantDepth**(const float depth)

Set constant depth value that is assumed if no depth measurement is passed using `accept(dv::measurements::Depth)`. By default the constant depth is assumed to be 3.0 meters, which is just a reasonable guess.

Parameters

depth – Distance to the scene (depth).

Throws

`InvalidArgument` – Exception is thrown if a negative depth value is passed.

inline *dv*::*EventStore* &**operator>>**(*dv*::*EventStore* &out)

Private Functions

inline *dv*::*kinematics*::*LinearTransformerf* **generateTransforms**(const int64_t from, const int64_t to)

Generate a sequence of transformations at a fixed period (`samplingPeriod`) with an additional overhead transform before and after the given interval.

Parameters

- **from** – Start of the interest interval.
- **to** – End of the interest interval.

Returns

Transformer with resampled transformations.

inline *dv*::*EventStore* **compensateEvents**(const *dv*::*EventStore* &events, const *dv*::*kinematics*::*LinearTransformerf* &transforms, const *dv*::*kinematics*::*Transformationf* &target, const float depth)

Apply motion compensation to event store and project all event into the target transformation.

Parameters

- **events** – Input events.
- **transforms** – Transformer containing the fine grained trajectory of the camera motion.
- **target** – Target position of the camera to be projected into.
- **depth** – Scene depth to be assumed for the calculations.

Returns

Motion compensated events at the target camera pose.

inline *dv*::*EventStore* **generateEventsAt**(const int64_t timestamp)

Generate compensated events at a given timestamp.

Parameters

timestamp – time to compensate events at.

Returns

A motion compensated events at given time point.

inline *dv*::*Frame* **generateFrameAt**(const int64_t timestamp)

Generate a frame at a given timestamp.

Parameters

timestamp – Time to generate frame at.

Returns

A motion compensated frame at given time point.

Private Members

PixelPredictor **predictor**

dv::*kinematics*::*LinearTransformerf* **transformer**

std::unique_ptr<*Accumulator*> **accumulator**

std::map<int64_t, float> **depths**

float **constantDepth** = 3.f

dv::*EventStore* **eventBuffer**

int64_t **storageDuration** = 5000000LL

const int64_t **samplingPeriod** = 200LL

MotionCompensator::Info **info**

template<class **MainStreamType**, class ...**AdditionalTypes**>

class **MultiStreamSlicer**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/multi_stream_slicer.hpp> *MultiStreamSlicer* takes multiple streams of timestamped data, slices data with configured intervals and calls a given callback method on each interval. It is an extension of *StreamSlicer* class that can synchronously slice multiple streams. Each stream has to be named uniquely, the name is carried over to the callback method to identify each stream.

The class relies heavily on templating, so it supports different containers of data, as long as the container is an iterable and each element contains an accessible timestamp in microsecond format.

The slicing is driven by the main stream, which needs to be specified during construction time. The type of the main stream is the first template argument and the name for the main stream is provided as the constructor's first argument.

By default, these types are supported without additional configuration: dv::EventStore, *dv::EventPacket*, *dv::TriggerPacket*, dv::cvector<dv::Trigger>, *dv::IMUPacket*, dv::cvector<dv::IMU>, dv::cvector<dv::Frame>. Additional types can be supported by specifying them as additional template parameters.

Template Parameters

- **MainStreamType** – The type of the main stream.
- **AdditionalTypes** – Parameter pack to specify an arbitrary number of additional stream types to be supported.

Public Types

using **InputType** = *std*::variant<*MainType*, *dv*::*EventStore*, *dv*::*EventPacket*, *dv*::*IMUPacket*, *dv*::*TriggerPacket*, *dv*::*cvector*<*dv*::*Frame*>, *dv*::*cvector*<*dv*::*IMU*>, *dv*::*cvector*<*dv*::*Trigger*>, *AdditionalTypes*...>

Alias for the variant that holds a packet type.

Public Functions

inline explicit **MultiStreamSlicer**(*std*::string mainStreamName)

Initialize the multi-stream slicer, provide the type of the main stream and a name for the main stream. The slicing is performed by applying a typical slicer on the main stream, all other stream follow it. When a window of slicing executes, the slicer extracts according data from all the other streams and calls a registered callback method for data processing.

Main stream is used to evaluate the jobs, but it also waits for the other types of data to arrive. The callbacks are not executed until all data has arrived on all streams.

By default, these types are supported without additional configuration: dv::EventStore, *dv::EventPacket*, *dv::TriggerPacket*, dv::cvector<dv::Trigger>, *dv::IMUPacket*, dv::cvector<dv::IMU>, dv::cvector<dv::Frame>. Additional types can be supported by specifying them as additional template parameters.

Parameters

mainStreamName – Name of the main stream.

template<class **DataType**>

inline void **addStream**(const *std*::string &streamName)

Add a stream to the slicer.

Template Parameters
DataType – Data packet type of the stream.

Parameters
streamName – Name for the stream.

template<class **DataType**>
inline void **accept**(const *std*::string &streamName, const *DataType* &data)

Accept incoming data for a stream and evaluate processing jobs. Can be either a packet or a single timestamped element of the stream.

Parameters

- **streamName** – Name of the stream.
- **data** – Incoming data, either a data packet or timestamp data element.

Throws
`RuntimeError` – Exception is thrown if passed data type does not match the stream data type.

inline int **doEveryTimeInterval**(const *dv*::*Duration* interval, *std*::function<void(const *dv*::*TimeWindow*&, const MapOfVariants&)> callback)

Register a callback to be performed at a given interval. Data is passed as an argument to the method. Callback method passes *TimeWindow* parameter along the data for the callback to be aware of time slicing windows.

Parameters

- **interval** – Interval at which the callback has to be executed.
- **callback** – Callback method that is called at the given interval, receives time window information and sliced data.

Returns
An id that can be used to modify this job.

inline int **doEveryTimeInterval**(const *dv*::*Duration* interval, *std*::function<void(const MapOfVariants&)> callback)

Register a callback to be performed at a given interval. Data is passed as an argument to the method.

Parameters

- **interval** – Interval at which the callback has to be executed.
- **callback** – Callback method that is called at the given interval.

Returns
An id that can be used to modify this job.

inline int **doEveryNumberOfElements**(const size_t n, *std*::function<void(const *dv*::*TimeWindow*&, const MapOfVariants&)> callback, const *TimeSlicingApproach* timeSlicingApproach = *TimeSlicingApproach*::*BACKWARD*)

Adds a number-of-elements triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback function every time `n` elements are added to the stream buffer, with the corresponding data. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Parameter timeSlicingApproach - is an enum that defines timing approach for multi-stream slicing by number. The slicing by number happens by slicing the main stream by a given number of elements. Secondary streams

are sliced by the time window of the numbered slice, this introduces a problem of gaps between two number slices - the gap values can either be assigned to current or the next slice, this enum allows to control which of the data parts these gap data will be assigned - backwards will assign all gap data from previous slice end time to current slice start time to current, the forwards approach will assign the gap data from current slice end time to next slice start time to the current slice. The forwards slice timing will result in processing delay of exactly one slice, as it requires to wait for the next slice to happen to correctly retrieve next slice start time. Backwards slicing does not wait for any additional data and processes everything immediately.

Parameters

- **n** – the interval (in number of elements) in which the callback should be called.
- **callback** – the callback function that gets called on the data every interval.
- **timeSlicingApproach** – Select approach for handling secondary stream gap data.

Returns
A handle to uniquely identify the job.

inline int **doEveryNumberOfElements**(const size_t n, *std*::function<void(const MapOfVariants&)> callback, const *TimeSlicingApproach* timeSlicingApproach = *TimeSlicingApproach*::*BACKWARD*)

Adds a number-of-elements triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback function every time `n` elements are added to the stream buffer, with the corresponding data. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Parameter timeSlicingApproach - is an enum that defines timing approach for multi-stream slicing by number. The slicing by number happens by slicing the main stream by a given number of elements. Secondary streams are sliced by the time window of the numbered slice, this introduces a problem of gaps between two number slices - the gap values can either be assigned to current or the next slice, this enum allows to control which of the data parts these gap data will be assigned - backwards will assign all gap data from previous slice end time to current slice start time to current, the forwards approach will assign the gap data from current slice end time to next slice start time to the current slice. The forwards slice timing will result in processing delay of exactly one slice, as it requires to wait for the next slice to happen to correctly retrieve next slice start time. Backwards slicing does not wait for any additional data and processes everything immediately.

Parameters

- **n** – the interval (in number of elements) in which the callback should be called.
- **callback** – the callback function that gets called on the data every interval.
- **timeSlicingApproach** – Select approach for handling secondary stream gap data.

Returns
A handle to uniquely identify the job.

inline void **modifyTimeInterval**(const int jobId, const *dv*::*Duration* timeInterval)

Modify the execution interval of a job.

Parameters

- **jobId** – Callback id that is received from callback registration.
- **timeInterval** – New time interval to be executed.

Throws
`invalid_argument` – Exception is thrown if trying to modify a number based slicing job.

inline void **modifyNumberInterval**(const int jobId, const size_t n)

Modify the execution number of elements of a job.

Parameters

- **jobId** – Job id that is received from callback registration.
- **n** – New number of elements to slice for the given job id.

Throws

`invalid_argument` – Exception is thrown if trying to modify a time based slicing job.

inline bool **hasJob**(const int jobId) const

Returns true if the slicer contains the slice-job with the provided id

Parameters

jobId – the id of the slice-job in question

Returns

true, if the slicer contains the given slice-job

inline void **removeJob**(const int jobId)

Removes the given job from the list of current jobs.

Parameters

jobId – The job id to be removed

inline void **setStreamSeekTime**(const *std*::string &streamName, const int64_t seekTimestamp)

Update a stream's seek time manually and evaluate jobs.

Data synchronization is automatically inferred from received data. This works well with data streams that produce data at guaranteed periodic intervals. For aperiodic data streams, which produce data spontaneously, a manual synchronization is required. This method allows to manually instruct the slicer that the given stream has provided data up to, but not including, this given seek timestamp; even in case when there was no data. Slicer is then able to progress other streams until the given time, since it assumes no data will ever arrive for this stream until this point. Be sure to call this method when you are sure no data will arrive, otherwise that data can be lost.

Parameters

- **streamName** – Name of the stream.
- **seekTimestamp** – Seek time for this stream; all data until this time has been provided to the slicer.

Protected Attributes

int64_t **mMainBufferSeekTime** = -1

Main buffer seek time, this is the timestamp of last fed data into main slicer.

std::map<int, SliceJob> **mSliceJobs**

Storage container for configured slice jobs.

int32_t **mHashCounter** = 0

std::map<int32_t, int32_t> **mMapFromSliceJobIdsToMainSlicerIds**

Map for determining mapping from multi stream slicer job ids to main stream slicer job ids, we use this since it is not known a priori how job ids are set for the main stream slicer

std::map<*std*::string, *InputType*> **mBuffer**

Buffered data that is in queue for slicing.

std::map<*std*::string, int64_t> **mLastReceivedBufferTimestamps**

Placeholder for manually provided seek timestamp of stream seek times.

std::string **mMainStreamName**

Name of the main stream.

dv::StreamSlicer<*MainStreamType*> **mMainSlicer**

Slicer for the main stream, all other streams follow the main stream slicer.

Private Types

using **MainType** = typename *std*::conditional_t<*dv*::*concepts*::*is_type_one_of*<*MainStreamType*, *dv*::*EventStore*, *dv*::*EventPacket*, *dv*::*IMUPacket*, *dv*::*TriggerPacket*, *dv*::*cvector*<*dv*::*Frame*>, *dv*::*cvector*<*dv*::*IMU*>, *dv*::*cvector*<*dv*::*Trigger*>, *AdditionalTypes*...>, *std*::monostate, *MainStreamType*>

Private Functions

inline int64_t **getMinLastBufferTimestamps**()

Get the minimum value of the last received buffer timestamps.

Returns
minimum last received buffer timestamp.

inline int64_t **getMinEvaluatedJobTime**()

Get the minimum of the last evaluated job times. This is helpful for determining which data to remove from the internal buffer as any data before this minimum value is no longer needed and can, therefore, be discarded

Returns
minimum of the last evaluated job times.

inline void **evaluate**()

Evaluate the current state of the slicer. Performs data book-keeping and executes the callback methods.

Private Static Functions

template<class **VectorType**>
static inline *VectorType* **sliceVector**(const int64_t start, const int64_t end, const *VectorType* &packet)

Slice a vector type within given time bounds [start, end). Start time is inclusive, end time is exclusive.

Template Parameters
VectorType –

Parameters

- **start** – Start timestamp
- **end** – End timestamp
- **packet** – Packet of a vector type

Returns
Copy of the data within the bounds

template<class **PacketType**>
static inline *PacketType* **slicePacketSpecific**(const int64_t start, const int64_t end, const *PacketType* &packet)

Templated method for packet slicing. Returns the data slice between given timestamps. Start time is inclusive, end time is exclusive.

Template Parameters
PacketType –

Parameters

- **start** – Start timestamp
- **end** – End timestamp
- **packet** – Packet of data

Returns
Copy of the data within the bounds

static inline *InputType* **slicePacket**(const int64_t start, const int64_t end, const *InputType* &packet)

Templated method for packet contained in a variant. Returns the data slice between given timestamps. Start time is inclusive, end time is exclusive.

Parameters

- **start** – Start of time range.
- **end** – End of time range.
- **packet** – Input data packet.

Returns
Sliced data from the packet according to given time ranges.

template<class **PacketType**>
static inline void **mergePackets**(const *PacketType* &from, *PacketType* &into)

Merge successive packets, this copies data from one to another. Performs shallow copy if possible.

Template Parameters
PacketType –

Parameters

- **from** – Source packet
- **into** – Destination packet

template<class **PacketType**>
static inline void **eraseUpToIterable**(const int64_t timeLimit, *PacketType* &packet)

Erase data within the packet up to the given time point. Specific implementation for vector containers.

Template Parameters
PacketType –

Parameters

- **timeLimit** – Timestamp to delete until, this is exclusive
- **packet** – Packet to modify

template<class **PacketType**>

static inline void **eraseUpTo**(const int64_t timeLimit, *PacketType* &packet)

Erase data within the packet up to the given time point.

Template Parameters
PacketType –

Parameters

- **timeLimit** – Timestamp to delete until, this is exclusive
- **packet** – Packet to modify

template<class **PacketType**>
static inline *dv*::*TimeWindow* **getPacketTimeWindow**(const *PacketType* &packet)

Retrieve highest and lowest timestamps of a given packet

Template Parameters
PacketType –

Parameters
packet –

Returns
Time window containing start and end timestamps.

template<class **PacketType**>
static inline bool **isPacketEmpty**(const *PacketType* &packet)

Check if a packet is empty.

Template Parameters
PacketType –

Parameters
packet –

Returns
True if the given packet is empty, false otherwise.

class **NetworkReader** : public *dv*::*io*::*CameraInputBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/network_reader.hpp> Network capture class. Connect to a TCP or a local socket server providing a data stream. The class provides a single data stream per network capture.

Public Functions

inline **NetworkReader**(const *std*::string_view ipAddress, const uint16_t port)

Initialize a network capture object, it will connect to a given TCP port with given IP address.

Parameters

- **ipAddress** – IP address of the target TCP server.
- **port** – TCP port number.

inline **NetworkReader**(const *std*::string_view ipAddress, const uint16_t port, boost::asio::ssl::context &&encryptionContext)

Initialize an encrypted network capture object, it will connect to a given TCP port with given IP address. Provide an encryption context that is preconfigured, prefer using existing `dv::io::encrypt::defaultEncryptionClient()` method for configuring the encryption context.

Parameters

- **ipAddress** – IP address of the target TCP server.
- **port** – TCP port number.
- **encryptionContext** – Preconfigured encryption context.

inline explicit **NetworkReader**(const *std*::filesystem::path &socketPath)

Initialize a network capture object, it will connect to a given UNIX socket with a given file system path.

Parameters

socketPath – Path to the UNIX socket.

inline virtual **~NetworkReader**()

Destructor - disconnects from network resource, stops threads and frees any buffered data.

inline virtual *std*::optional<*dv*::*EventStore*> **getNextEventBatch**() override

Read next event batch. This is a non-blocking method, if there is no data to read, it will return a `std::nullopt`.

Returns

Next batch of events, `std::nullopt` if no data received from last read or the event stream is not available.

inline virtual *std*::optional<*dv*::*Frame*> **getNextFrame**() override

Read next frame. This is a non-blocking method, if there is no data to read, it will return a `std::nullopt`.

Returns

Next frame, `std::nullopt` if no data received from last read or the event stream is not available.

inline virtual *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **getNextImuBatch**() override

Read next imu measurement batch. This is a non-blocking method, if there is no data to read, it will return a `std::nullopt`.

Returns

Next batch of imu measurements, `std::nullopt` if no data received from last read or the event stream is not available.

inline virtual *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **getNextTriggerBatch**() override

Read next trigger batch. This is a non-blocking method, if there is no data to read, it will return a `std::nullopt`.

Returns

Next batch of triggers, `std::nullopt` if no data received from last read or the event stream is not available.

inline virtual *std*::optional<cv::Size> **getEventResolution**() const override

Retrieve the event sensor resolution. The method returns `std::nullopt` if event stream is not available or the metadata does not contain resolution.

Returns

Event sensor resolution or `std::nullopt` if not available.

inline virtual *std*::optional<cv::Size> **getFrameResolution**() const override

Retrieve the frame sensor resolution. The method returns `std::nullopt` if frame stream is not available or the metadata does not contain resolution.

Returns

Frame sensor resolution or `std::nullopt` if not available.

template<class **PacketType**>
inline *std*::shared_ptr<*PacketType*> **getNextPacket**()

Read next packet, given it's type.

The given type must match the stream type exactly (it must be a flatbuffer generated type). Returns `nullptr` if no data is available for reading or stream of such type is not available.

Template Parameters
PacketType – *Stream* packet type, must be a flatbuffer type and must match stream type exactly.

Returns
Shared pointer to a packet of data, or nullptr if unavailable.

inline virtual bool **isEventStreamAvailable**() const override

Check whether an event stream is available in this capture class.

Returns
True if an event stream is available; false otherwise.

inline virtual bool **isFrameStreamAvailable**() const override

Check whether a frame stream is available in this capture class.

Returns
True if a frame stream is available; false otherwise.

inline virtual bool **isImuStreamAvailable**() const override

Check whether an *IMU* data stream is available in this capture class.

Returns
True if an *IMU* data stream is available; false otherwise.

inline virtual bool **isTriggerStreamAvailable**() const override

Check whether a trigger stream is available in this capture class.

Returns
True if a trigger stream is available; false otherwise.

inline virtual *std*::string **getCameraName**() const override

Get camera name, which is a combination of the camera model and the serial number.

Returns
String containing the camera model and serial number separated by an underscore character.

inline virtual bool **isRunning**() const override

Check whether the network stream is still connected.

Returns
True if network stream is running and available.

template<class **PacketType**>
inline bool **isStreamAvailable**() const

Check whether a stream of given type is available.

The given type must match the stream type exactly (it must be a flatbuffer generated type). Returns `nullptr` if no data is available for reading or stream of such type is not available.

Template Parameters
PacketType – *Stream* packet type, must be a flatbuffer type and must match stream type exactly.

Returns
True if stream of a given type is available, false otherwise.

inline void **close**()

Explicitly close the communication socket, receiving data is not going to possible after this method call.

inline const *dv*::*io*::*Stream* &**getStreamDefinition**() const

Get the stream definition object, which describe the available data stream by this reader.

Returns
Data stream definition object.

Private Types

using **PacketQueue** = boost::lockfree::spsc_queue<*dv*::*types*::*TypedObject**>

Private Functions

inline void **readClbk**(*std*::vector<*std*::byte> &data, const int64_t)

Read block of data from the network socket.

Parameters
data – Container for data that is going to be read.

inline void **connectTCP**(const *std*::string_view ipAddress, const uint16_t port, const bool tlsEnabled = false)

Initiate connection to the given IP address and port.

Parameters

- **ipAddress** – Ip address, dot separated (in format "0.0.0.0")
- **port** – TCP port number
- **tlsEnabled** – Enable TLS encryption

inline void **connectUNIX**(const *std*::filesystem::path &socketPath)

Initiate a connection to UNIX socket under given filesystem path.

Parameters
socketPath – Path to a socket.

inline void **readThread**()

inline void **initializeReader**()

Private Members

std::function<void(*std*::vector<*std*::byte>&, const int64_t)> **mReadHandler** = *std*::bind_front(&*NetworkReader*::*readClbk*, this)

Callback method that calls read method of the socket.

boost::asio::io_service **mIOService**

IO service context.

std::unique_ptr<*network*::*SocketBase*> **mSocket** = nullptr

Socket to contain the connection instance.

asioSSL::context **mTLSContext** = asioSSL::context(asioSSL::context::method::tlsv12_client)

Decryption context.

bool **mTLSEnabled**

Whether TLS encryption is enabled.

dv::*io*::*Reader* **mAedat4Reader**

AEDAT4 reader.

dv::*io*::*Stream* **mStream**

Data stream container - one per capture.

std::string **mCameraName**

Name of the camera producing the stream.

PacketQueue **mPacketQueue** = *PacketQueue*(1000)

Incoming packet queue.

std::thread **mReadingThread**

Reading thread.

std::atomic<bool> **mKeepReading** = true

Atomic bool used to stop the reading thread.

std::atomic<bool> **mExceptionThrown** = false

Boolean value that indicated whether an exception was thrown on reading thread.

std::exception_ptr **mException** = nullptr

Pointer that holds thrown exception, mExceptionThrown contains thread-safe flag indicating an exception was thrown

class **NetworkWriter** : public *dv*::*io*::*CameraOutputBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/network_writer.hpp> Network server class for streaming AEDAT4 serialized data types.

Public Types

using **ErrorMessageCallback** = *std*::function<void(const boost::system::error_code&, const *std*::string_view)>

Public Functions

inline **NetworkWriter**(const *std*::string_view ipAddress, const uint16_t port, const *dv*::*io*::*Stream* &stream, const size_t maxClientConnections = 10, *ErrorMessageCallback* messageCallback = `[](const boost::system::error_code &, const std::string_view) { })`

Create a non-encrypted server that listens for connections on a given IP address. Supports multiple clients.

Parameters

- **ipAddress** – IP address to bind the server.
- **port** – Port number.
- **stream** – AEDAT4 stream definition.
- **maxClientConnections** – Maximum number of client connections supported by this instance.
- **messageCallback** – Callback to handle any error messages received by the client connections.

inline **NetworkWriter**(const *std*::string_view ipAddress, const uint16_t port, const *dv*::*io*::*Stream* &stream, boost::asio::ssl::context &&encryptionContext, const size_t maxClientConnections = 10, *ErrorMessageCallback* messageCallback = `[](const boost::system::error_code &, const std::string_view) { })`

Create an encrypted server that listens for connections on a given IP address. Supports multiple clients.

Parameters

- **ipAddress** – IP address to bind the server.
- **port** – Port number.
- **stream** – AEDAT4 stream definition.
- **encryptionContext** – Preconfigured encryption context, use either `dv::io::encrypt::defaultEncryptionServer()` to create the context or configure custom encryption context. When a client connects to the server, it will run handshake, during which client certificates will be validated, if the handshake fails, connection is terminated.
- **maxClientConnections** – Maximum number of client connections supported by this instance.
- **messageCallback** – Callback to handle any error messages received by the client connections.

inline **NetworkWriter**(const *std*::filesystem::path &socketPath, const *dv*::*io*::*Stream* &stream, const size_t maxClientConnections = 10, *ErrorMessageCallback* messageCallback = `[](const boost::system::error_code &, const std::string_view) { })`

Create a local socket server. Provide a path to the socket, if a file already exists on a given path, the connection will fail by throwing an exception. It is required that the given socket path does not point to an existing socket

file. If the file *can* exist, it is up to the user of this class to decide whether it is safe to remove any existing socket files or the class should not bind to the path.

Parameters

- **socketPath** – Path to a socket file, must be a non-existent path.
- **stream** – AEDAT4 stream definition.
- **maxClientConnections** – Maximum number of client connections supported by this instance.
- **messageCallback** – Callback to handle any error messages received by the client connections.

inline virtual **~NetworkWriter**()

Closes the socket, frees allocated memory, and removes any queued packets from write queue.

inline virtual void **writeEvents**(const *EventStore* &events) override

Write an event store to the network stream.

Parameters

events – Data to be sent out.

inline virtual void **writeFrame**(const *dv*::*Frame* &frame) override

Write a frame image to the network stream.

Parameters

frame – Data to be sent out.

inline virtual void **writeIMU**(const *cvector*<*dv*::*IMU*> &imu) override

Write *IMU* data to the socket.

Parameters

imu – Data to be sent out.

inline virtual void **writeTriggers**(const *cvector*<*dv*::*Trigger*> &triggers) override

Write trigger data to the network stream.

Parameters

triggers – Data to be sent out.

template<class **PacketType**>
inline void **writePacket**(*PacketType* &&packet)

Write a flatbuffer packet to the network stream.

Template Parameters

PacketType – Type of the packet, must satisfy the *dv::concepts::FlatbufferPacket* concept.

Parameters

packet – Data to write.

inline virtual *std*::string **getCameraName**() const override

Get camera name. It is looked up from the stream definition during construction.

Returns

inline size_t **getQueuedPacketCount**() const

Get number of packets in the write queue.

Returns

Number of packets in the write queue.

inline size_t **getClientCount**()

Get number of active connected clients.

Returns
Number of active connected clients.

Private Types

using **WriteQueue** = boost::lockfree::spsc_queue<*std*::shared_ptr<*dv*::*types*::*TypedObject*>>

Private Functions

template<class **SocketType**>
inline void **acceptStart**()

inline void **writePacketToClients**(const *std*::shared_ptr<*dv*::*types*::*TypedObject*> &packet)

inline void **ioThread**()

inline void **connectTCP**(const *std*::string_view ipAddress, const uint16_t port)

inline void **connectUNIX**(const *std*::filesystem::path &socketPath)

inline void **generateHeaderContent**(const *dv*::*io*::*Stream* &stream)

inline void **removeClient**(const *Connection* *const client)

Private Members

std::string **mCameraName**

size_t **mMaxConnections**

asio::io_service **mIoService**

std::unique_ptr<asioTCP::acceptor> **mAcceptorTcp** = nullptr

std::unique_ptr<asioTCP::socket> **mAcceptorTcpSocket** = nullptr

std::unique_ptr<asioUNIX::acceptor> **mAcceptorUnix** = nullptr

std::unique_ptr<asioUNIX::socket> **mAcceptorUnixSocket** = nullptr

asioSSL::context **mTLSContext** = asioSSL::context(asioSSL::context::method::tlsv12_server)

bool **mTLSEnabled**

std::mutex **mClientsMutex**

std::vector<*Connection**> **mClients**

The client list is raw point, that is self-owned, read Connection class documentation for more details.

std::atomic<size_t> **mQueuedPackets** = 0

dv::*io*::*Writer* **mAedat4Writer**

dv::*cstring* **mInfoNode**

std::atomic<bool> **mShutdownRequested** = false

std::thread **mIOThread**

int32_t **mStreamId** = 0

std::filesystem::path **mSocketPath**

WriteQueue **mWriteQueue** = *WriteQueue*(1024)

ErrorMessageCallback **mErrorMessageHandler**

Error message handler, by default: NOOP.

class **NoneCompressionSupport** : public *dv*::*io*::*compression*::*CompressionSupport*

Public Functions

inline explicit **NoneCompressionSupport**(const *CompressionType* type)

inline virtual void **compress**(*dv*::*io*::*support*::*IODataBuffer* &packet) override

class **NoneDecompressionSupport** : public *dv*::*io*::*compression*::*DecompressionSupport*

Public Functions

inline explicit **NoneDecompressionSupport**(const *CompressionType* type)

inline virtual void **decompress**(*std*::vector<*std*::byte> &source, *std*::vector<*std*::byte> &target) override

class **NoRedetection** : public *dv*::*features*::*RedetectionStrategy*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/redetection_strategy.hpp> No redetection strategy.

Public Functions

inline virtual bool **decideRedetection**(const *TrackerBase*&) override

Do not perform redetection.

Returns
Just return false always.

struct **NullPointer** : public *dv*::*exceptions*::*info*::*EmptyException*

struct **Observation** : public *flatbuffers*::NativeTable

Public Types

typedef *ObservationFlatbuffer* **TableType**

Public Functions

inline **Observation**()

inline **Observation**(int32_t _trackId, int32_t _cameraId, const *dv*::*cstring* &_cameraName, int64_t _timestamp)

Public Members

int32_t **trackId**

int32_t **cameraId**

dv::*cstring* **cameraName**

int64_t **timestamp**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

struct **ObservationBuilder**

Public Functions

inline void **add_trackId**(int32_t trackId)

inline void **add_cameraId**(int32_t cameraId)

inline void **add_cameraName**(*flatbuffers*::Offset<*flatbuffers*::String> cameraName)

inline void **add_timestamp**(int64_t timestamp)

inline explicit **ObservationBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

ObservationBuilder &**operator=**(const *ObservationBuilder*&)

inline *flatbuffers*::Offset<*ObservationFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **ObservationFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *Observation* **NativeTableType**

Public Functions

inline int32_t **trackId**() const

The tracking sequence ID that the landmark is observed by a camera.

inline int32_t **cameraId**() const

Arbitrary ID of the camera, this can be application specific.

inline const *flatbuffers*::String ***cameraName**() const

Name of the camera. Optional.

inline int64_t **timestamp**() const

Timestamp of the observation (µs).

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *Observation* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*Observation* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*Observation* *_o, const *ObservationFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*ObservationFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Observation* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

template<typename **_Scalar**, int **NX** = Eigen::Dynamic, int **NY** = Eigen::Dynamic>

class **OptimizationFunctor**

#include </builds/inivation/dv/dv-processing/include/dv-processing/optimization/optimization_functor.hpp> Basic functor class inherited by all contrastMaximization functor. This functor is used by Eigen/NumericalDiff class, which handles the non linear optimization underlying contrast maximization algorithm. For more information about contrast maximization please check "contrast_maximization_rotation.hpp" or "contrast_maximization_translation_and_depth.hpp".

Template Parameters

- **_Scalar** – type of variable to optimize (e.g. int, float..).
- **NX** – Number of input variables (note: all variables are stored as Nx1 vector of values)
- **NY** – Number of output measurements (note: number of measurements needs to be at least as big as number of input variables - NX - otherwise the optimization problem cannot be solved.)

Public Types

Values:

enumerator **InputsAtCompileTime**

enumerator **ValuesAtCompileTime**

typedef *_Scalar* **Scalar**

typedef Eigen::Matrix<*Scalar*, InputsAtCompileTime, 1> **InputType**

typedef Eigen::Matrix<*Scalar*, ValuesAtCompileTime, 1> **ValueType**

typedef Eigen::Matrix<*Scalar*, ValuesAtCompileTime, InputsAtCompileTime> **JacobianType**

Public Functions

virtual int **operator()**(const Eigen::VectorXf &input, Eigen::VectorXf &cost) const = 0

Base method for cost function implementation.

Parameters

- **input** – parameters to be optimized
- **cost** – cost value updated at each iteration of the optimization.

Returns

optimization result (positive if successful)

inline **OptimizationFunctor**(int inputs, int values)

Constructor for cost optimization parameters

Parameters

- **inputs** – number of inputs to be optimized
- **values** – number of functions evaluation for gradient computation

inline int **inputs**() const

getter for size of input parameters to be optimized.

Returns

number of input parameters optimized.

inline int **values**() const

getter for size of function evaluations performed at each optimization iteration.

Returns

number of function evaluations at each optimization iteration.

Private Members

int **mInputs**

int **mValues**

struct **optimizationOutput**

Public Members

int **optimizationSuccessful**

int **iter**

Eigen::VectorXf **optimizedVariable**

struct **optimizationParameters**

Public Members

float **learningRate** = float(1e-1)

float **epsfcn** = 0

float **ftol** = 0.000345267

float **gtol** = 0

float **xtol** = 0.000345267

int **maxfev** = 400

struct **OutOfRange** : public *dv*::*exceptions*::*info*::*EmptyException*

struct **OutputError**

Public Types

using **Info** = *ErrorInfo*

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

template<*concepts*::*AddressableEvent* **EventType**, class **EventPacketType**>

class **PartialEventData**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> **INTERNAL USE ONLY** Internal event container class that holds a shard of events. A `PartialEventData` holds a shared pointer to an `EventPacket`, which is the underlying data structure. The underlying data can either be const, in which case no addition is allowed, or non const, in which addition of new data is allowed. Slicing is allowed in both cases, as it only modifies the control structure. All the events in the partial have to be monotonically increasing in time. A `PartialEventData` can be sliced both from the front as well as from the back. By doing so, the memory footprint of the structure is not modified, just the internal bookkeeping pointers are readjusted. The `PartialEventData` keeps track of lowest as well as highest times of events in the structure.

The data `PartialEventData` points to can be shared between multiple `PartialEventData`, each with potentially different slicings.

Public Functions

inline explicit **PartialEventData**(const size_t capacity = 10000)

Creates a new *PartialEventData* shard. Allocates new memory on the heap to keep the data. Upon constructions, the newly created object is the sole owner of the data.

Parameters
capacity – Number of events this data partial can store.

inline explicit **PartialEventData**(*std*::shared_ptr<const *EventPacketType*> data)

Creates a new *PartialEventData* shard from existing const data. Copies the supplied shared_ptr into the structure, acquiring shared ownership of the supplied data.

Parameters
data – The shared pointer to the data to which we want to obtain shared ownership

PartialEventData(const *PartialEventData* &other) = default

Copy constructor. Creates a shallow copy of `other` without copying the actual data over. As slicing does not alter the underlying data, the new copy may be sliced without affecting the orignal object.

Parameters
other –

inline *iterator* **iteratorAtTime**(const int64_t time) const

Returns an iterator to the first element that is bigger than the supplied timestamp. If every element is bigger than the supplied time, an iterator to the first element is returned (same as *begin()*). If all elements have a smaller timestamp than the supplied, the end iterator is returned (same as *end()*).

Parameters
time – The requested time. The iterator will be the first element with a timestamp larger than this time.

Returns
An iterator to the first element larger than the supplied time.

inline *iterator* **begin**() const

Returns an iterator to the first element of the *PartialEventData*. The iterator is according to the current slice and not to the underlying datastore. E.g. when slicing the shard from the front, the *begin()* will change.

Returns
Returns an iterator at the beginning data partial

inline *iterator* **end**() const

Returns an iterator to one after the last element of the *PartialEventData*. The iterator is according to the current slice and not to the underlying datastore. E.g. when slicing the shard from the back, the result of *end()* will change.

Returns
Returns an iterator at the end of the data partial

inline void **sliceFront**(const size_t number)

Slices off `number` events from the front of the *PartialEventData*. This operation just adjust the bookkeeping of the datastructure without actually modifying the underlying data representation. If there are not enough events left, a `range_error` exception is thrown.

Other instances of *PartialEventData* which share the same underlying data are not affected by this.

Parameters
number – amount of events to be removed from the front.

inline void **sliceBack**(const size_t number)

Slices off `number` events from the back of the *PartialEventData*. This operation just adjust the bookkeeping of the datastructure without actually modifying the underlying data representation. If there are not enough events left, a `range_error` exception is thrown.

Other instances of *PartialEventData* which share the same underlying data are not affected by this.

Parameters
number – amount of events to be removed from the back.

inline size_t **sliceTimeFront**(const int64_t time)

Slices off all the events that occur before the supplied time. The resulting data structure has a `lowestTime > time` where time is the supplied time.

This operation just adjust the bookkeeping of the datastructure without actually modifying the underlying data representation. If there are not enough events left, a `range_error` exception is thrown.

Other instances of *PartialEventData* which share the same underlying data are not affected by this.

Parameters
time – the threshold time. All events `<= time` will be sliced off

Returns
number of events that actually got sliced off as a result of this operation.

inline size_t **sliceTimeBack**(const int64_t time)

Slices off all the events that occur after the supplied time. The resulting data structure has a `lowestTime < time` where time is the supplied time.

This operation just adjust the bookkeeping of the datastructure without actually modifying the underlying data representation. If there are not enough events left, a `range_error` exception is thrown.

Other instances of *PartialEventData* which share the same underlying data are not affected by this.

Parameters
time – the threshold time. All events `> time` will be sliced off

Returns
number of events that actually got sliced off as a result of this operation.

inline void **_unsafe_addEvent**(const *EventType* &event)

UNSAFE OPERATION Copies the data of the supplied event into the underlying data structure and updates the internal bookkeeping to accommodate the event.

NOTE: This function does not perform any boundary checks. Any call to function is expected to have performed the following boundary checks: *canStoreMoreEvents()* to see if there is space to accommodate the new event. *getHighestTime()* has to be smaller or equal than the new event's timestamp, as we require events to be monotonically increasing.

Parameters
event – The event to be added

inline void **_unsafe_moveEvent**(*EventType* &&event)

UNSAFE OPERATION Moves the data of the supplied event into the underlying data structure and updates the internal bookkeeping to accommodate the event.

NOTE: This function does not perform any boundary checks. Any call to function is expected to have performed the following boundary checks: *canStoreMoreEvents()* to see if there is space to accommodate the new event. *getHighestTime()* has to be smaller or equal than the new event's timestamp, as we require events to be monotonically increasing.

Parameters
event – The event to be added

inline *EventType* &**front**()

Get a reference to the first available event in the partial.

Returns
Reference to first element in the partial.

inline *EventType* &**back**()

Get a reference to the last available event in the partial.

Returns
Reference to last element in the partial.

inline size_t **getLength**() const

The length of the current slice of data. This value can be in range [0; capacity].

Returns
the current length of the slice in number of events.

inline int64_t **getLowestTime**() const

Gets the lowest timestamp of an event that is represented in this Partial. The lowest timestamp is always identical to the timestamp of the first event of the slice.

Returns
The timestamp of the first event in the slice. This is also the lowest time present in this slice.

inline int64_t **getHighestTime**() const

Gets the highest timestamp of an event that is represented in this Partial. The lowest timestamp is always identical to the timestamp of the last event of the slice.

Returns
The timestamp of the last event in the slice. This is also the highest timestamp present in this slice.

inline const *EventType* &**operator[]**(size_t offset) const

Returns a reference to the element at the given offset of the slice.

Parameters
offset – The offset in the slice of which element a reference should be obtained

Returns
A reference to the object at offset offset

inline bool **canStoreMoreEvents**() const

Checks if it is safe to add more events to this partial. It is safe to add more events when the following conditions are fulfilled:

- The partial does not represent const data. In that case, any modification of the underlying buffer is impossible.
- The partial does not exceed the sharding count limit
- The partial hasn't been sliced from the back

If it has been sliced from the back, adding new events would put them in unreachable space.

Returns
true if there is space available to store more events in this partial.

inline size_t **availableCapacity**() const

Amount of space still available in this data partial.

Returns
Amount of events this data partial can store additionally.

inline bool **merge**(const *PartialEventData* &other)

Merge the other data partial into this one by copying the contents, if that is possible. If merge is not possible, the function returns false and does nothing.

Parameters
other – Other data partial to be merged into this one.

Returns
True if merge was successful, false otherwise.

Private Types

using **iterator** = typename *dv*::*cvector*<const *EventType*>::iterator

Private Members

bool **referencesConstData_**

size_t **start_**

size_t **length_**

size_t **capacity_**

int64_t **lowestTime_**

int64_t **highestTime_**

std::shared_ptr<*EventPacketType*> **modifiableDataPtr_**

std::shared_ptr<const *EventPacketType*> **data_**

Friends

friend class dv::io::MonoCameraWriter

friend class dv::io::NetworkWriter

template<*concepts*::*AddressableEvent* **EventType**, class **EventPacketType**>

class **PartialEventDataTimeComparator**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> **INTERNAL USE ONLY**
Comparator Functor that checks if a given time lies within bounds of the event packet

Public Functions

inline explicit **PartialEventDataTimeComparator**(const bool lower)

inline bool **operator()**(const *PartialEventData*<*EventType*, *EventPacketType*> &partial, const int64_t time) const

Returns true, if the comparator is set to not lower and the given time is higher than the highest timestamp of the partial, or when it is set to lower and the timestamp is higher than the lowest timestamp of the partial.

Parameters

- **partial** – The partial to be analysed
- **time** – The time to be compared against

Returns
true, if time is higher than either lowest or highest timestamp of partial depending on state

inline bool **operator()**(const int64_t time, const *PartialEventData*<*EventType*, *EventPacketType*> &partial) const

Returns true, if the comparator is set to not lower and the given time is higher than the lowest timestamp of the partial, or when it is set to lower and the timestamp is higher than the highest timestamp of the partial.

Parameters

- **partial** – The partial to be analysed
- **time** – The time to be compared against

Returns
true, if time is higher than either lowest or lowest timestamp of partial depending on state

Private Members

const bool **lower_**

struct **PixelDisparity**

#include </builds/inivation/dv/dv-processing/include/dv-processing/depth/sparse_event_block_matcher.hpp> Structure containing disparity results for a point of interest.

Public Functions

inline **PixelDisparity**(const cv::Point2i &coordinates, const bool valid, const *std*::optional<float> correlation = *std*::nullopt, const *std*::optional<float> score = *std*::nullopt, const *std*::optional<int32_t> disparity = *std*::nullopt, const *std*::optional<cv::Point2i> &templatePosition = *std*::nullopt, const *std*::optional<cv::Point2i> &matchedPosition = *std*::nullopt)

Initialize the disparity structure.

Parameters

- **coordinates** – Point of interest coordinates, this will contain same coordinates that were passed into the algorithm.
- **valid** – Holds true if the disparity match valid. False otherwise.

- **correlation** – Pearson correlation value for the best matching block, if available. This value is in the range [-1.0; 1.0].
- **score** – Matching score value, if available. This value is in the range [0.0; 1.0].
- **disparity** – Disparity value in pixels, if available. The value is in the range [minDisparity; maxDisparity].
- **templatePosition** – Requested coordinate of interest point in the left (rectified) image pixel space.
- **matchedPosition** – Best match coordinate on the right (rectified) image pixel space.

Public Members

cv::Point2i **coordinates**

Point of interest coordinates, this will contain same coordinates that were passed into the algorithm.

bool **valid**

Holds true if the disparity match valid. False otherwise.

std::optional<float> **correlation**

Pearson correlation value for the best matching block, if available. This value is in the range [-1.0; 1.0]. Correlation value of -1.0 will mean that matched patch is an inverse of the original template patch, 1.0 will be an equal match, 0.0 is no correlation. A positive value indicates a positive correlation between searched template patch and best match, which could be considered a good indication of a correct match.

std::optional<float> **score**

Standard score (Z-score) for the match, if available. The score is the number of standard deviations the highest probability value is above the mean of all probabilities of the matching method.

std::optional<int32_t> **disparity**

Disparity value in pixels, if available. The value is in the range [minDisparity; maxDisparity].

std::optional<cv::Point2i> **templatePosition**

Coordinates of the matching template on the left (rectified) image space. Set to `std::nullopt` if the template coordinates are out-of-bounds.

std::optional<cv::Point2i> **matchedPosition**

Coordinates of the matched template on the right (rectified) image space. Set to `std::nulltopt` if a match cannot be reliably found, otherwise contains coordinates with highest correlation match on the right side rectified camera pixel space.

class **PixelMotionPredictor**

Public Types

using **SharedPtr** = *std*::shared_ptr<*PixelMotionPredictor*>

using **UniquePtr** = *std*::unique_ptr<*PixelMotionPredictor*>

Public Functions

inline explicit **PixelMotionPredictor**(const *camera*::CameraGeometry::SharedPtr &cameraGeometry)

Construct pixel motion predictor class.

Parameters

cameraGeometry – Camera geometry class instance containing intrinsic calibration of the camera sensor.

virtual **~PixelMotionPredictor**() = default

inline *dv*::*EventStore* **predictEvents**(const *dv*::*EventStore* &events, const *Transformationf* &dT, const float depth) const

Apply delta transformation to event input and generate new transformed event store with new events that are within the new camera perspective (after applying delta transform).

Parameters

- **events** – Input events.
- **dT** – Delta transformation to be applied.
- **depth** – Scene depth.

Returns

Transformed events.

template<*concepts*::*Coordinate2DMutableIterable* **Output**, *concepts*::*Coordinate2DIterable* **Input**>
inline *Output* **predictSequence**(const *Input* &points, const *Transformationf* &dT, const float depth) const

Apply delta transformation to coordinate input and generate new transformed coordinate array with new coordinates that are within the new camera perspective (after applying delta transform).

Parameters

- **points** – Input coordinate array.
- **dT** – Delta transformation to be applied.
- **depth** – Scene depth.

Returns

Transformed point coordinates.

template<*concepts*::*Coordinate2DCostructible* **Output**, *concepts*::*Coordinate2D* **Input**>
inline *Output* **predict**(const *Input* &pixel, const *Transformationf* &dT, const float depth) const

Reproject given pixel coordinates using the delta transformation and depth.

Parameters

- **pixel** – Input pixel coordinates.
- **dT** – Delta transformation.
- **depth** – Scene depth.

Returns
Transformed pixel coordinate using the delta transform, camera geometry and scene depth.

inline bool **isUseDistortion**() const

Is the distortion model enabled for the reprojection of coordinates.

Returns
True if the distortion model is enabled, false otherwise.

inline void **setUseDistortion**(bool useDistortion_)

Enable of disable the usage of a distortion model.

Parameters
useDistortion_ – Pass true to enable usage of the distortion model, false otherwise.

Private Members

const *dv*::*camera*::*CameraGeometry*::*SharedPtr* **camera**

bool **useDistortion** = false

struct **Pose** : public *flatbuffers*::NativeTable

Public Types

typedef *PoseFlatbuffer* **TableType**

Public Functions

inline **Pose**()

inline **Pose**(int64_t _timestamp, const Vec3f &_translation, const Quaternion &_rotation, const *dv*::*cstring* &_referenceFrame, const *dv*::*cstring* &_targetFrame)

Public Members

int64_t **timestamp**

Vec3f **translation**

Quaternion **rotation**

dv::*cstring* **referenceFrame**

dv::*cstring* **targetFrame**

Public Static Functions

static inline constexpr const char *__GetFullyQualifiedName__()

Friends

inline friend *std*::ostream &__operator<<__(*std*::ostream &os, const *Pose* &packet)

struct __PoseBuilder__

Public Functions

inline void __add_timestamp__(int64_t timestamp)

inline void __add_translation__(const Vec3f *translation)

inline void __add_rotation__(const Quaternion *rotation)

inline void __add_referenceFrame__(*flatbuffers*::Offset<*flatbuffers*::String> referenceFrame)

inline void __add_targetFrame__(*flatbuffers*::Offset<*flatbuffers*::String> targetFrame)

inline explicit __PoseBuilder__(*flatbuffers*::FlatBufferBuilder &_fbb)

PoseBuilder &__operator=__(const *PoseBuilder*&)

inline *flatbuffers*::Offset<*PoseFlatbuffer*> __Finish__()

Public Members

flatbuffers::FlatBufferBuilder &__fbb___

flatbuffers::uoffset_t __start___

struct __PoseFlatbuffer__ : private *flatbuffers*::Table

#include </builds/inivation/dv/dv-processing/include/dv-processing/data/pose_base.hpp> A struct holding timestamp and pose.

Public Types

typedef *Pose* __NativeTableType__

Public Functions

inline int64_t **timestamp**() const

Timestamp (µs).

inline const Vec3f ***translation**() const

Translational vector.

inline const Quaternion ***rotation**() const

Rotation quaternion.

inline const *flatbuffers*::String ***referenceFrame**() const

Name of the reference frame (transforming from)

inline const *flatbuffers*::String ***targetFrame**() const

Name of the target frame (transforming into)

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *Pose* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*Pose* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*Pose* *_o, const *PoseFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*PoseFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Pose* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "POSE"

class **PoseVisualizer**

#include </builds/inivation/dv/dv-processing/include/dv-processing/visualization/pose_visualizer.hpp> Visualize the current and past poses as an image.

Public Types

enum class **Mode**

Values:

enumerator **CUSTOM**

enumerator **VIEW_XY**

enumerator **VIEW_YZ**

enumerator **VIEW_ZX**

enumerator **VIEW_XZ**

enumerator **VIEW_YX**

enumerator **VIEW_ZY**

enum class **GridPlane**

Values:

enumerator **PLANE_NONE**

enumerator **PLANE_XY**

enumerator **PLANE_YZ**

enumerator **PLANE_ZX**

Public Functions

inline explicit EIGEN_MAKE_ALIGNED_OPERATOR_NEW **PoseVisualizer**(const size_t trajectoryLength = 10000, const cv::Size2i &resolution = cv::Size2i(640, 480))

Constructor.

Parameters

resolution – size of the generated images in pixels

inline void **updateCameraPosition**(const Eigen::Vector3f &newPosition)

Update the position in which camera is located.

Parameters

newPosition – New translational position of the camera in world coordinate frame.

inline void **setViewMode**(const *Mode* mode)

Set the mode in which the pose viewer will be working.

Parameters
mode – New viewing mode

inline void **setViewMode**(const *std*::string &str)

inline void **setGridPlane**(const *GridPlane* plane)

Set the plane on which the grid will be displayed.

Parameters
plane – Grid plane

inline void **setGridPlane**(const *std*::string &str)

inline void **updateCameraOrientation**(const float yawDeg, const float pitchDeg, const float rollDeg)

Update the orientation of the camera expressed as XYZ Euler angles.

Parameters

- **yawDeg** – Camera yaw in degrees
- **pitchDeg** – Camera pitch in degrees
- **rollDeg** – Camera roll in degrees

inline void **setFrameSize**(const cv::Size2i &newSize)

Update the size of output image.

Parameters
newSize – New output image dimensions.

inline void **setCoordinateDimensions**(const float newSize)

Update the displayed coordinate frame size.

Parameters
newSize – [m]

inline void **setLineThickness**(const int newThickness)

Update the line thickness of the drawing.

Parameters
newThickness – Drawing line thickness in pixels.

inline void **accept**(const *dv*::*LandmarksPacket* &landmarks)

Add markers for drawing.

Parameters
landmarks – A packet of landmarks to be drawn on the preview.

inline void **accept**(const *dv*::*kinematics*::*Transformationf* &pose)

Add a new pose to the visualization.

Parameters
pose – New pose for visualization.

inline int64_t **getTimestamp**() const

Return the timestamp of the most recent pose.

Returns
Timestamp in Unix microsecond format.

inline *dv*::*Frame* **generateFrame**()

Return a visualization image.

Returns
The generated image.

inline void **reset**()

Reset the pose history and set an offset to the last pose.

inline const cv::Scalar &**getBackgroundColor**() const

Get the background color.

Returns
Background color.

inline void **setBackgroundColor**(const cv::Scalar &backgroundColor)

Set new background color.

Parameters
backgroundColor – OpenCV scalar for the background color.

inline const cv::Scalar &**getGridColor**() const

Get the grid line color.

Returns
Grid line color

inline void **setGridColor**(const cv::Scalar &gridColor)

Set new grid line color

Parameters
mGridColor – OpenCV scalar for the grid line color.

inline bool **getDrawLinesToLandmarks**() const

Check whether drawing of lines to landmark markers is enabled.

Returns
True if drawing of lines is enabled, false otherwise.

inline void **setDrawLinesToLandmarks**(bool drawLinesToLandmarks)

Enable or disable drawing of lines from camera to active landmarks. Active landmarks are those which were accepted by the visualizer with last `accept(dv::LandmarksPacket)` call.

Parameters
drawLinesToLandmarks –

inline size_t **getLandmarkLimit**() const

Get the maximum number of landmarks to be drawn.

Returns
Maximum number of landmarks

inline void **setLandmarkLimit**(size_t numLandmarks)

Set a limit for number of landmarks that are stored and drawn.

Parameters
numLandmarks – Number of landmarks

inline size_t **getLandmarkSize**() const

Get the number of landmarks currently stored in the visualizer.

Returns
Number of landmarks stored in the visualizer.

inline void **clearLandmarks**()

Remove all landmarks stored in the landmarks buffer.

Private Functions

inline cv::Point2f **projectPose**(const Eigen::Vector4f &pose_W, const Eigen::Vector4f &mask = Eigen::Vector4f(1.f, 1.f, 1.f, 1.f)) const

Convert a pose from 3D coordinates to image frame.

Parameters

- **pose_W** – pose to project in the World frame
- **mask** – Mask will be applied as a component-wise multiplication on the pose

Returns
Projected pose coordinates

inline void **refreshCameraMatrix**()

Update the camera matrix based on the current image size.

inline void **initMinMax**()

Initialize minimum and maximum point coordinates.

Private Members

cv::Scalar **mBackgroundColor** = cv::Scalar(30, 30, 30)

cv::Scalar **mGridColor** = cv::Scalar(128, 128, 128)

cv::Size2i **mResolution**

int **mLineThickness** = 1

GridPlane **mGridPlane** = *GridPlane*::*PLANE_ZX*

Eigen::Vector4f **mMinPoint_W**

Eigen::Vector4f **mMaxPoint_W**

float **mFrameSize** = 1.0

boost::circular_buffer<Eigen::Vector4f, Eigen::aligned_allocator<Eigen::Vector4f>> **mPath**

dv::*kinematics*::*LinearTransformerf* **mTrajectory**

std::map<int64_t, *Marker*> **mMarkers**

size_t **mMarkerLimit** = 10'000

bool **mDrawLinesToMarker** = true

std::vector<int64_t> **mTimestamps**

dv::*kinematics*::*Transformationf* **mLastPose**

int64_t **mLastTimestamp** = 0

Eigen::Vector3f **mCameraPosition**

Eigen::Quaternionf **mCameraOrientation**

const float **mFocalLength** = 100

Eigen::Matrix<float, 3, 3> **mCamMat**

Eigen::Matrix<float, 4, 4> **mT_CW**

Mode **mViewMode** = *Mode*::*VIEW_ZX*

dv::*kinematics*::*Transformationf* **mT_OW**

Private Static Functions

static inline int **getGridSpan**(const float maxSpan)

Calculate the optimal grid span based on the maximum position span and the user defined density.

Parameters

maxSpan – Maximum arbitrary span

Returns

Optimal span value

class **Reader**

Public Types

using **ReadHandler** = *dv*::*std_function_exact*<void(*std*::vector<*std*::byte>&, const int64_t)>

Public Functions

inline explicit **Reader**(*dv*::*io*::*support*::*TypeResolver* resolver = *dv*::*io*::*support*::*defaultTypeResolver*, *std*::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> stats = nullptr)

~Reader() = default

Reader(const *Reader* &other) = delete

Reader &**operator=**(const *Reader* &other) = delete

Reader(*Reader* &&other) noexcept = default

Reader &**operator=**(*Reader* &&other) noexcept = default

inline void **verifyVersion**(const *ReadHandler* &readHandler)

inline *std*::unique_ptr<const *dv*::*IOHeader*> **readHeader**(const *ReadHandler* &readHandler)

inline *std*::unique_ptr<const *dv*::*FileDataTable*> **readFileDataTable**(const uint64_t size, const int64_t position, const *ReadHandler* &readHandler)

inline *std*::tuple<*dv*::PacketHeader, *std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **readPacket**(const *ReadHandler* &readHandler)

inline *std*::tuple<*dv*::PacketHeader, *std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **readPacket**(const int64_t byteOffset, const *ReadHandler* &readHandler)

inline *dv*::PacketHeader **readPacketHeader**(const *ReadHandler* &readHandler)

inline *dv*::PacketHeader **readPacketHeader**(const int64_t byteOffset, const *ReadHandler* &readHandler)

inline *std*::pair<*std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **readPacketBody**(const *dv*::*FileDataDefinition* &packet, const *ReadHandler* &readHandler)

inline *std*::pair<*std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **readPacketBody**(const int32_t streamId, const uint64_t size, const *ReadHandler* &readHandler)

inline *std*::pair<*std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **readPacketBody**(const int32_t streamId, const uint64_t size, const int64_t byteOffset, const *ReadHandler* &readHandler)

inline *std*::unique_ptr<const *dv*::*FileDataTable*> **buildFileDataTable**(const uint64_t fileSize, const *ReadHandler* &readHandler)

inline *std*::vector<*dv*::*io*::*Stream*> **getStreams**() const

inline *CompressionType* **getCompressionType**() const

Private Functions

inline void **readFromInput**(const uint64_t length, const int64_t position, const *ReadHandler* &readHandler)

inline void **decompressData**()

Private Members

dv::*io*::*support*::*TypeResolver* **mTypeResolver**

std::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> **mStats**

std::unique_ptr<*dv*::*io*::*compression*::*DecompressionSupport*> **mDecompressionSupport**

std::vector<*std*::byte> **mReadBuffer**

std::vector<*std*::byte> **mDecompressBuffer**

std::unordered_map<int32_t, *dv*::*io*::*Stream*> **mStreams**

Private Static Functions

static inline *std*::unique_ptr<const *dv*::*IOHeader*> **decodeHeader**(const *std*::vector<*std*::byte> &header)

static inline *std*::unique_ptr<const *dv*::*FileDataTable*> **decodeFileDataTable**(const *std*::vector<*std*::byte> &table)

static inline *std*::unique_ptr<*dv*::*types*::*TypedObject*> **decodePacketBody**(const *std*::vector<*std*::byte> &packet, const *dv*::*types*::*Type* &type)

class **ReadOnlyFile** : private *dv*::*io*::*SimpleReadOnlyFile*

Public Functions

ReadOnlyFile() = delete

inline explicit **ReadOnlyFile**(const *std*::filesystem::path &filePath, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*, *std*::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> stats = nullptr)

inline const auto &**getFileInfo**() const

inline *std*::vector<*std*::pair<*std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*>> **read**(const int64_t startTimestamp, const int64_t endTimestamp, const int32_t streamId)

Return all packets containing data with timestamps between a given start and end timestamp, meaning all data with a timestamp in [start, end].

Parameters

- **startTimestamp** – start timestamp of range, inclusive.
- **endTimestamp** – end timestamp of range, inclusive.
- **streamId** – data stream ID (separate logical type).

Returns

packets containing data within given timestamp range.

inline *std*::pair<*std*::unique_ptr<*dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **read**(const *dv*::*FileDataDefinition* &packet)

inline *std*::pair<*std*::unique_ptr<const *dv*::*types*::*TypedObject*>, const *dv*::*io*::*support*::*Sizes*> **read**(const int32_t streamId, const uint64_t size, const int64_t byteOffset)

Public Static Functions

static inline bool **inRange**(const int64_t rangeStart, const int64_t rangeEnd, const *dv*::*FileDataDefinition* &packet)

static inline bool **aheadOfRange**(const int64_t rangeStart, const int64_t rangeEnd, const *dv*::*FileDataDefinition* &packet)

static inline bool **pastRange**(const int64_t rangeStart, const int64_t rangeEnd, const *dv*::*FileDataDefinition* &packet)

Private Functions

inline void **parseHeader**()

inline void **loadFileDataTable**()

inline void **readClbk**(*std*::vector<*std*::byte> &data, const int64_t byteOffset)

inline void **createFileInfo**()

Private Members

dv::*io*::*FileInfo* **mFileInfo**

dv::*io*::*Reader* **mReader**

Private Static Functions

static inline *dv*::*cvector*<*dv*::*FileDataDefinition*>::const_iterator **getStartingPointForTimeRangeSearch**(const int64_t startTimestamp, const *dv*::*FileDataTa* &streamDataTable)

class **RedetectionStrategy**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/redetection_strategy.hpp> Implementation of different redetection strategies for trackers.

Subclassed by *dv::features::FeatureCountRedetection*, *dv::features::NoRedetection*, *dv::features::UpdateIntervalOrFeatureCountRedetection*, *dv::features::UpdateIntervalRedetection*

Public Types

typedef *std*::shared_ptr<*RedetectionStrategy*> **SharedPtr**

typedef *std*::unique_ptr<*RedetectionStrategy*> **UniquePtr**

Public Functions

virtual bool **decideRedetection**(const *dv*::*features*::*TrackerBase* &tracker) = 0

Decide the redetection of tracker features depending on the state of the tracker.

Parameters
tracker – Current state of the tracker.

Returns
True to perform redetection of features, false to continue.

inline bool **decideRedection**(const *dv*::*features*::*TrackerBase* &tracker)

Decide the redetection of tracker features depending on the state of the tracker.

Deprecated:

Use decideRedetection instead

Parameters
tracker – Current state of the tracker.

Returns
True to perform redetection of features, false to continue.

virtual **~RedetectionStrategy**() = default

template<class **EventStoreClass** = *dv*::*EventStore*>

class **RefractoryPeriodFilter** : public *dv*::*EventFilterBase*<*dv*::*EventStore*>

Public Functions

inline explicit **RefractoryPeriodFilter**(const cv::Size &resolution, const *dv*::*Duration* refractoryPeriod = *dv*::*Duration*(250))

Refractory period filter discards any events that are registered at a pixel location that already had an event within the refractory period. Refractory period should be relatively small value (in the range of one or a few hundred microseconds).

Parameters

- **resolution** – Sensor resolution.
- **refractoryPeriod** – Refractory period duration.

inline virtual bool **retain**(const typename *EventStoreClass*::value_type &event) noexcept override

Test whether event satisfies (is larger than) refractory period test.

Parameters
event – Event to be tested.

Returns
True - there were no events within the refractory period at that pixel location, false otherwise.

inline *RefractoryPeriodFilter* &**operator<<**(const *EventStoreClass* &events)

Accept events using the input stream operator.

Parameters
events – Input events.

Returns

inline *dv*::*Duration* **getRefractoryPeriod**() const

Get the refractory period.

Returns
Currently configured refractory period.

inline void **setRefractoryPeriod**(const *dv*::*Duration* refractoryPeriod)

Set a new refractory period value.

Parameters
refractoryPeriod – New refractory period value.

Private Members

dv::*TimeSurface* **mTimeSurface**

int64_t **mRefractoryPeriod**

struct **Result**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/tracker_base.hpp> *Result* of tracking.

Public Types

typedef *std*::shared_ptr<*Result*> **SharedPtr**

typedef *std*::shared_ptr<const *Result*> **ConstPtr**

Public Functions

inline **Result**(const int64_t _timestamp, const *dv*::*cvector*<*dv*::*TimedKeyPoint*> &_keypoints, const bool keyframe)

Construct tracking result

Parameters

- **_timestamp** – Execution time of tracking
- **_keypoints** – The resulting features
- **keyframe** – Whether this set of features can be regarded as a keyframe (redetection was triggered)

Result() = default

Public Members

dv::*cvector*<*dv*::*TimedKeyPoint*> **keypoints** = {}

A vector of keypoints.

bool **asKeyFrame** = false

A flag that notifies the user of the tracker that this specific input caused redetection to happen and the tracker not only tracked the buffered events, but also detected new features.

int64_t **timestamp** = 0

Timestamp of the execution, it can be frame timestamp or last timestamp of an event slice.

class **RotationIntegrator**

Public Functions

inline explicit **RotationIntegrator**(const *dv*::*kinematics*::*Transformationf* &T_S_target = *dv*::*kinematics*::*Transformationf*(), int64_t sensorToTargetTimeOffset = 0, const Eigen::Vector3f &gyroscopeOffset = {0.f, 0.f, 0.f})

Parameters

- **T_S_target** – initial target position wrt to sensor
- **sensorToTargetTimeOffset** – temporal offset between sensor (imu) and target. t_target = t_sensor - offset
- **gyroscopeOffset** – constant measurement offset in gyroscope samples [radians].

inline Eigen::Matrix3f **getRotation**() const

Getter outputting current target transformation relative to (target) initial one

Returns
[3x3] rotation matrix

inline void **setT_S_target**(const *dv*::*kinematics*::*Transformationf* &T_S_target)

Setter to update target position wrt to the sensor

Parameters
T_S_target – new target transformation wrt sensor

inline int64_t **getTimestamp**() const

Getter outputting timestamp of current target transformation

Returns
timestamp

inline *dv*::*kinematics*::*Transformation*<float> **getTransformation**() const

Getter returning [4x4] transformation corresponding to current target position wrt (target) initial one

Returns
4x4 transformation corresponding to current integrated rotation

inline void **accept**(const *dv*::*IMU* &imu)

Update sensor position with new measurement

Parameters
imu – single imu measurement

Private Functions

inline Eigen::Matrix3f **rotationMatrixFromImu**(const *dv*::*IMU* &imu, const float dt)

Transform gyroscope measurement into rotation matrix representation

Parameters
imu – single imu measurement

Returns
[3x3] rotation matrix corresponding to rotation measured from gyroscope

Private Members

Eigen::Matrix4f **mT_S0_target**

matrix storing target position wrt to the sensor (imu)

int64_t **mSensorToTargetTimeOffset**

offset [us] between sensor and target: t_target = t_sensor - offset

Eigen::Vector3f **mGyroscopeOffset**

measurement offset [radians] along each x, y, z axis of the sensor

Eigen::Matrix3f **mR_S0_S** = Eigen::Matrix3f::Identity(3, 3)

matrix storing current sensor orientation wrt initial sensor orientation

int64_t **mTimestamp** = -1

timestamp of current sensor position wrt initial time.

class **RotationLossFunctor** : public *dv*::*optimization*::*OptimizationFunctor*<float>

#include </builds/inivation/dv/dv-processing/include/dv-processing/optimization/contrast_maximization_rotation.hpp>
Given a chunk of events, the idea of contrast maximization is to warp events in space and time given a predefined motion model. Contrast maximization aims at finding the optimal parameters of the given motion model. The idea is that if the motion is perfectly estimated, all events corresponding to the same point in the scene, will be warped to the same image plane location, at a given point in time. If this happens, the reconstructed event image will be sharp, having high contrast. This high contrast is measured as variance in the image. For this reason, contrast maximization searches for the best motion parameters which maximize the contrast of the event image reconstructed after warping events in space to a spacific point in time. In order to warp event in space and time we use the "dv::kinematics::MotionCompensator" class. This contrast maximization class assumes pure camera rotational motion model. Given a set of imu samples and events in a time range, gyroscope measurement offset if optimized. The gyroscope offset is optimized instead of each single gyroscope measurement in order to limit the search space of the non linear optimization. In addition, given the high sample rate of imu, it would be hard to achieve real time computing optimizing each single gyroscope value. For this reason, the gyroscope offset (x, y, z) is optimized and assumed to be constant among all the gyroscope samples.

Public Functions

inline **RotationLossFunctor**(*dv*::*camera*::*CameraGeometry*::*SharedPtr* &camera, const *dv*::*EventStore* &events, float contribution, const *dv*::*cvector*<*dv*::*IMU*> &imuSamples, const *dv*::*kinematics*::*Transformationf* &T_S_target, int64_t imuToCamTimeOffsetUs, int inputDim, int numMeasurements)

This contrast maximization class assumes pure camera rotational motion model. Given a set of imu samples and events in a time range, gyroscope measurement offset if optimized. The gyroscope offset is optimized instead of each single gyroscope measurement in order to limit the search space of the non linear optimization.

Parameters

- **camera** – Camera geometry used to create motion compensator
- **events** – Events used to compute motion compensated image
- **contribution** – Contribution value of each event to the total pixel intensity
- **imuSamples** – Chunk of imu samples used to compensate events. These values (gyrosvcope part) are updated with the gyroscope measurement offset, which is the optimized variable.
- **T_S_target** – Transformation from sensor (imu) to target (camera). Used to convert imu motion into camera motion.
- **imuToCamTimeOffsetUs** – Time synchronization offset between imu and camera
- **inputDim** – Number of parameters to optimize
- **numMeasurements** – Number of function evaluation performed to compute the gradient

inline virtual int **operator()**(const Eigen::VectorXf &gyroscopeOffsetImu, Eigen::VectorXf &stdInverse) const

Implementation of the objective function: optimize gyroscope offset. Current cost is stored in stdInverse. Notice that since we want to maximize the contrast but optimizer minimize cost function we use as cost 1/contrast

Private Members

dv::*camera*::*CameraGeometry*::*SharedPtr* **mCamera**

Camera geometry data. This information is used to create motionCompensator and compensate events.

const *dv*::*EventStore* **mEvents**

Raw events compensated using imu data.

float **mContribution**

Event contribution for total pixel intensity. This parameter is very important since it strongly influence contrast value. It needs to be tuned based on scene and length of event chunk.

const *dv*::*cvector*<*dv*::*IMU*> **mImuSamples**

Imu data used to compensate mEvents.

const *dv*::*kinematics*::*Transformationf* **mT_S_target**

Target (i.e. camera) to imu transformation. Used to construct rotationIntegrator that keeps track of camera position.

int64_t **mImuToTargetTimeOffsetUs**

Time offset between imu and target. Check rotationIntegrator class for more information.

struct **RuntimeError** : public *dv*::*exceptions*::*info*::*EmptyException*

template<*dv*::*concepts*::*EventToFrameConverter*<*dv*::*EventStore*> **AccumulatorClass** = *dv*::*EdgeMapAccumulator*>

class **SemiDenseStereoMatcher**

#include *</builds/inivation/dv/dv-processing/include/dv-processing/depth/semi_dense_stereo_matcher.hpp>*
Semi-dense stereo matcher - a class that performs disparity calculation using an OpenCV dense disparity calculation algorithm. The implementation performs accumulation of a stereo pair of images of input events and applies the given stereo disparity matcher algorithm (semi-global block matching by default).

Public Functions

inline **SemiDenseStereoMatcher**(*std*::unique_ptr<*AccumulatorClass*> leftAccumulator, *std*::unique_ptr<*AccumulatorClass*> rightAccumulator, const *std*::shared_ptr<cv::StereoMatcher> &matcher = cv::StereoSGBM::create())

Construct a semi dense stereo matcher object by providing custom accumulators for left and right cameras and a stereo matcher class.

Parameters

- **leftAccumulator** – *Accumulator* for the left camera.
- **rightAccumulator** – *Accumulator* for the right camera.
- **matcher** – Stereo matcher algorithm, if not provided, the implementation will use a cv::StereoSGBM class with default parameters.

inline explicit **SemiDenseStereoMatcher**(const cv::Size &leftResolution, const cv::Size &rightResolution, const *std*::shared_ptr<cv::StereoMatcher> &matcher = cv::StereoSGBM::create())

Construct a semi dense stereo matcher with default accumulator settings and a stereo matcher class.

Parameters

- **leftResolution** – Resolution of the left camera.
- **rightResolution** – Resolution of the right camera.
- **matcher** – Stereo matcher algorithm, if not provided, the implementation will use a cv::StereoSGBM class with default parameters.

inline explicit **SemiDenseStereoMatcher**(*std*::unique_ptr<*dv*::*camera*::*StereoGeometry*> geometry, *std*::shared_ptr<cv::StereoMatcher> matcher = *dv*::*depth*::*defaultStereoMatcher*())

Construct a semi dense stereo matcher with default accumulator settings and a stereo matcher class. The calibration information about camera will be extracted from the stereo geometry class instance.

Parameters

- **geometry** – Object describing the stereo camera geometry.
- **matcher** – Stereo matcher algorithm, if not provided, the implementation will use a cv::StereoSGBM class with optimized parameters.

inline **SemiDenseStereoMatcher**(*std*::unique_ptr<*dv*::*camera*::*StereoGeometry*> geometry,
std::unique_ptr<*AccumulatorClass*> leftAccumulator,
std::unique_ptr<*AccumulatorClass*> rightAccumulator,
std::shared_ptr<cv::StereoMatcher> matcher =
dv::*depth*::*defaultStereoMatcher*())

Construct a semi dense stereo matcher object by providing custom accumulators for left and right cameras and a stereo matcher class. The calibration information about camera will be extracted from the stereo geometry class instance.

Parameters

- **geometry** – Object describing the stereo camera geometry.
- **leftAccumulator** – *Accumulator* for the left camera.
- **rightAccumulator** – *Accumulator* for the right camera.
- **matcher** – Stereo matcher algorithm, if not provided, the implementation will use a cv::StereoSGBM class with optimized parameters.

inline cv::Mat **computeDisparity**(const *dv*::*EventStore* &left, const *dv*::*EventStore* &right)

Compute disparity of the two given event stores. The events will be accumulated using the accumulators for left and right camera accordingly and disparity is computed using the configured block matching algorithm. The function is not going to slice the input events, so event streams have to be synchronized and sliced accordingly. The `dv::StereoEventStreamSlicer` class is a good option for slicing stereo event streams.

NOTE: Accumulated frames will be rectified only if a stereo geometry class was provided during construction.

See also:

dv::StereoEventStreamSlicer for synchronized slicing of a stereo event stream.

Parameters

- **left** – Events from left camera.
- **right** – Events from right camera.

Returns
Disparity map computed by the configured block matcher.

inline cv::Mat **compute**(const cv::Mat &leftImage, const cv::Mat &rightImage) const

Compute stereo disparity given a time synchronized pair of images. Images will be rectified before computing disparity if a StereoGeometry class instance was provided.

Parameters

- **leftImage** – Left image of a stereo pair of images.
- **rightImage** – Right image of a stereo pair of images.

Returns
Disparity map computed by the configured block matcher.

inline const *dv*::*Frame* &**getLeftFrame**() const

Retrieve the accumulated frame from the left camera event stream.

Returns
An accumulated frame.

inline const *dv*::*Frame* &**getRightFrame**() const

Retrieve the accumulated frame from the right camera event stream.

Returns
An accumulated frame.

inline *dv*::*DepthEventStore* **estimateDepth**(const cv::Mat &disparity, const *dv*::*EventStore* &events, const float disparityScale = 16.f) const

Estimate depth given the disparity map and a list of events. The coordinates will be rectified and a disparity value will be looked up in the disparity map. The depth of each event is calculated using an equation: depth = (focalLength * baseline) / (disparity * pixelPitch). Focal length is expressed in meter distance.

The function requires knowledge about the pixel pitch distance which needs to be provided prior to calculations. The pixel pitch can be available in the camera calibration (in this case it will be looked up during construction of the class). If the pixel pitch is not available there, it must be provided manually using the `setPixelPitch` method. The pixel pitch value can be looked up in *dv::io::CameraCapture* class in case if running the stereo estimation in a live camera scenario.

For practical applications, depth estimation should be evaluated prior to any use. The directly estimated depth values can contain measurable errors which should be accounted for - the errors can usually be within 10-20% fixed absolute error distance. Usually this comes from various inaccuracies and can be mitigated by introducing a correction factor for the depth estimate.

Parameters

- **disparity** – Disparity map.
- **events** – Input events.
- **disparityScale** – Scale of disparity value in the disparity map, if subpixel accuracy is enabled in the block matching, this value will be equal to 16.

Returns
A depth event store, the events will contain the same information as in the input, but additionally will have the depth value. Events whose coordinates are outside of image bounds after rectification will be skipped.

inline *dv*::*DepthFrame* **estimateDepthFrame**(const cv::Mat &disparity, const float disparityScale = 16.f) const

Convert a disparity map into a depth frame. Each disparity value is converted into depth using the equation depth = (focalLength * baseline) / (disparity * pixelPitch). Output frame contains distance values expressed in integer values of millimeter distance.

Parameters

- **disparity** – Input disparity map.
- **disparityScale** – Scale of disparity value in the disparity map, if subpixel accuracy is enabled in the block matching, this value will be equal to 16.

Returns
A converted depth frame.

Protected Attributes

std::shared_ptr<cv::StereoMatcher> **mMatcher** = nullptr

std::unique_ptr<*AccumulatorClass*> **mLeftAccumulator** = nullptr

std::unique_ptr<*AccumulatorClass*> **mRightAccumulator** = nullptr

dv::*Frame* **mLeftFrame**

dv::*Frame* **mRightFrame**

std::unique_ptr<*dv*::*camera*::*StereoGeometry*> **mStereoGeometry** = nullptr

Private Functions

inline void **validateStereoGeometry**() const

Validates stereo geometry pointer, throws an error if the value is unset.

class **SimpleFile**

Subclassed by *dv::io::SimpleReadOnlyFile*, *dv::io::SimpleWriteOnlyFile*

Public Functions

constexpr **SimpleFile**() = default

inline explicit **SimpleFile**(const *std*::filesystem::path &filePath, const *ModeFlags* modeFlags, const *WriteFlags* writeFlags = *WriteFlags*::*NONE*, const size_t bufferSize = 65536)

Open a file for reading and/or writing, supporting extra modes for writing and buffer control. Will always do what you expect and throw an exception if there's any issue.

Parameters

- **filePath** – file path to open.
- **modeFlags** – Open file for reading, writing or both.
- **writeFlags** – If opening for writing, extra flags for truncation and append modes.
- **bufferSize** – Size of user-space buffer for file operations. Default 64KB, use 0 to disable buffering entirely.

inline **~SimpleFile**() noexcept

SimpleFile(const *SimpleFile* &file) = delete

SimpleFile &**operator=**(const *SimpleFile* &rhs) = delete

inline **SimpleFile**(*SimpleFile* &&file) noexcept

inline *SimpleFile* &**operator=**(*SimpleFile* &&rhs) noexcept

inline bool **isOpen**() const

inline void **flush**()

inline void **write**(const *std*::string_view data)

template<typename **T**>
inline void **write**(const *std*::vector<*T*> &data)

template<typename **T**>
inline void **write**(const *dv*::*cvector*<*T*> &data)

template<typename **T**>
inline void **write**(const *T* *elem, size_t num)

template<typename **S**, typename ...**Args**>
inline void **format**(const *S* &format, *Args*&&... args)

inline void **read**(*std*::string &data) const

template<typename **T**>
inline void **read**(*std*::vector<*T*> &data) const

inline void **read**(*dv*::*cstring* &data) const

template<typename **T**>
inline void **read**(*dv*::*cvector*<*T*> &data) const

template<typename **T**>
inline void **read**(*T* *elem, size_t num) const

inline void **readAll**(*std*::string &data) const

inline void **readAll**(*std*::vector<uint8_t> &data) const

inline void **readAll**(*dv*::*cstring* &data) const

inline void **readAll**(*dv*::*cvector*<uint8_t> &data) const

inline uint64_t **tell**() const

inline void **seek**(const uint64_t offset, const *SeekFlags* flags = *SeekFlags*::*START*) const

inline void **rewind**() const

inline uint64_t **fileSize**() const

inline *std*::filesystem::path **path**() const

Private Functions

inline void **close**() noexcept

Private Members

std::FILE ***f** = {nullptr}

char ***fBuffer** = {nullptr}

std::filesystem::path **fPath** = {}

class **SimpleReadOnlyFile** : private *dv*::*io*::*SimpleFile*

Subclassed by *dv::io::ReadOnlyFile*

Public Functions

constexpr **SimpleReadOnlyFile**() = default

inline explicit **SimpleReadOnlyFile**(const *std*::filesystem::path &filePath, const size_t bufferSize = 65536)

inline uint64_t **fileSize**() const

inline bool **isOpen**() const

inline *std*::filesystem::path **path**() const

inline void **read**(*std*::string &data) const

template<typename **T**>
inline void **read**(*std*::vector<*T*> &data) const

inline void **read**(*dv*::*cstring* &data) const

template<typename **T**>
inline void **read**(*dv*::*cvector*<*T*> &data) const

template<typename **T**>
inline void **read**(*T* *elem, size_t num) const

inline void **readAll**(*std*::string &data) const

inline void **readAll**(*std*::vector<uint8_t> &data) const

inline void **readAll**(*dv*::*cstring* &data) const

inline void **readAll**(*dv*::*cvector*<uint8_t> &data) const

inline void **rewind**() const

inline void **seek**(const uint64_t offset, const *SeekFlags* flags = *SeekFlags*::*START*) const

inline uint64_t **tell**() const

class **SimpleWriteOnlyFile** : private *dv*::*io*::*SimpleFile*

Subclassed by *dv::io::WriteOnlyFile*

Public Functions

constexpr **SimpleWriteOnlyFile**() = default

inline explicit **SimpleWriteOnlyFile**(const *std*::filesystem::path &filePath, const *WriteFlags* writeFlags = *WriteFlags*::*NONE*, const size_t bufferSize = 65536)

inline uint64_t **fileSize**() const

inline void **flush**()

template<typename **S**, typename ...**Args**>
inline void **format**(const *S* &format, *Args*&&... args)

inline bool **isOpen**() const

inline *std*::filesystem::path **path**() const

inline void **rewind**() const

inline void **seek**(const uint64_t offset, const *SeekFlags* flags = *SeekFlags*::*START*) const

inline uint64_t **tell**() const

inline void **write**(const *std*::string_view data)

template<typename **T**>
inline void **write**(const *std*::vector<*T*> &data)

template<typename **T**>
inline void **write**(const *dv*::*cvector*<*T*> &data)

template<typename **T**>
inline void **write**(const *T* *elem, size_t num)

struct **Sizes**

Public Members

uint64_t **mPacketElements** = {0}

uint64_t **mPacketSize** = {0}

uint64_t **mDataSize** = {0}

class **SliceJob**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/multi_stream_slicer.hpp> Internal container of slice jobs.

Public Types

enum class **SliceType**

Values:

enumerator **NUMBER**

enumerator **TIME**

using **JobCallback** = *std*::function<void(const *dv*::*TimeWindow*&, const *MapOfVariants*&)>

Callback method signature alias.

Public Functions

inline **SliceJob**(const int64_t intervalUS, *JobCallback* callback)

Create a slice job

Parameters

- **intervalUS** – Job execution interval in microseconds
- **callback** – The callback method

inline **SliceJob**(const size_t number, const *TimeSlicingApproach* slicing, *JobCallback* callback)

Create a slice by number job

Parameters

- **number** – Number of elements to be sliced
- **slicing** – Slicing method for gaps between numeric slices
- **callback** – The callback method

inline void **run**(const *dv*::*TimeWindow* &timeWindow, const *MapOfVariants* &data)

Public Members

SliceType **mType**

JobCallback **mCallback**

The callback method.

int64_t **mInterval** = -1

Job execution interval in microseconds.

size_t **mNumberOfElements** = 0

Slice by number configuration value.

TimeSlicingApproach **mTimeSlicing** = *TimeSlicingApproach*::*BACKWARD*

Time slicing method for slicing by number.

int64_t **mLastEvaluatedTimestamp** = 0

Timestamp specifying the last timestamp the job evaluated over.

class **SliceJob**

INTERNAL USE ONLY A single job of the EventStreamSlicer

Public Types

enum class **SliceType**

Values:

enumerator **NUMBER**

enumerator **TIME**

Public Functions

inline **SliceJob**(const *SliceType* type, const int64_t timeInterval, const size_t numberInterval, *std*::function<void(const *dv*::*TimeWindow*&, PacketType&)> callback)

INTERNAL USE ONLY Creates a new SliceJob of a certain type, interval and callback

Parameters

- **type** – The type of periodicity. Can be either NUMBER or TIME
- **timeInterval** – The interval at which the job should be executed
- **numberInterval** – The interval at which the job should be executed
- **callback** – The callback function to call on execution.

SliceJob() = default

inline void **run**(const PacketType &packet)

INTERNAL USE ONLY This function establishes how much fresh data is availble and how often the callback can be executed on this fresh data. it then creates slices of the data and executes the callback as often as possible.

Parameters

packet – the storage packet to slice on.

inline void **setTimeInterval**(const int64_t timeInterval)

INTERNAL USE ONLY Sets the time interval to the supplied value

Parameters

timeInterval – the new time interval to use

inline void **setNumberInterval**(const size_t numberInterval)

INTERNAL USE ONLY Sets the number interval to the supplied value

Parameters

numberInterval – the new interval to use

Public Members

size_t **mLastCallEnd** = 0

Private Members

SliceType **mType** = *SliceType*::*TIME*

const *std*::function<void(const *TimeWindow*&, PacketType&)> **mCallback**

int64_t **mTimeInterval** = 0

size_t **mNumberInterval** = 0

int64_t **mLastCallEndTime** = 0

Private Static Functions

template<class **ElementVector**>
static inline *ElementVector* **sliceByNumber**(const *ElementVector* &packet, const size_t fromIndex, const size_t number)

template<class **ElementVector**>
static inline *ElementVector* **sliceByTime**(const *ElementVector* &packet, const int64_t start, const int64_t end, size_t &endIndex)

class **SocketBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/network/socket_base.hpp> Interface class to define a socket API.

Subclassed by *dv::io::network::TCPTLSSocket*, *dv::io::network::UNIXSocket*

Public Types

using **CompletionHandler** = *std*::function<void(const boost::system::error_code&, const size_t)>

Callback alias that is used to handle a completed IO operation.

Public Functions

virtual **~SocketBase**() = default

virtual bool **isOpen**() const = 0

Check whether a socket is open and active.

Returns
True if socket is open, false otherwise.

virtual void **close**() = 0

Close the underlying socket communication. Async reads/writes can be aborted during this function call.

virtual void **write**(const asio::const_buffer &buffer, *CompletionHandler* &&handler) = 0

Write a data buffer to the socket asynchronously. Completion handler is called when write to the socket is complete.

Parameters

- **buffer** – Data buffer to written to the socket.
- **handler** – Completion handler, that is called when write is complete.

virtual void **read**(const asio::mutable_buffer &buffer, *CompletionHandler* &&handler) = 0

Read a data buffer from the socket asynchronously. Completion handler is called when read from the socket is complete.

Parameters

- **buffer** – Output buffer to place data from the socket.
- **wrHandler** – Completion handler, that is called when write is complete.

virtual void **syncWrite**(const asio::const_buffer &buffer) = 0

Write data into the socket synchronously, this method is a blocking call which returns when writing data is complete.

Parameters
buffer – Data to be written into the socket.

virtual void **syncRead**(const asio::mutable_buffer &buffer) = 0

Read data from the socket synchronously, this method is a blocking call which returns when reading data is complete.

Parameters
buffer – Output buffer to place data from the socket.

struct **SortedPacketBuffers**

Public Functions

inline void **acceptPacket**(const *std*::shared_ptr<libcaer::events::EventPacket> &packet)

inline void **clearBuffers**()

inline *std*::optional<*dv*::*EventStore*> **popEvents**(const int64_t timeOffset)

inline *std*::optional<*dv*::*Frame*> **popFrame**(const int64_t timeOffset)

inline *std*::optional<*dv*::*cvector*<*dv*::*IMU*>> **popImu**(const int64_t timeOffset)

inline *std*::optional<*dv*::*cvector*<*dv*::*Trigger*>> **popTriggers**(const int64_t timeOffset)

Public Members

size_t **packetCount** = 0

boost::lockfree::spsc_queue<*EventPacketPair*, boost::lockfree::capacity<10000>> **events**

boost::lockfree::spsc_queue<*EventPacketPair*, boost::lockfree::capacity<10000>> **imu**

boost::lockfree::spsc_queue<*EventPacketPair*, boost::lockfree::capacity<10000>> **triggers**

boost::lockfree::spsc_queue<*EventPacketPair*, boost::lockfree::capacity<1000>> **frames**

int64_t **eventStreamSeek** = -1

int64_t **imuStreamSeek** = -1

int64_t **triggerStreamSeek** = -1

int64_t **framesStreamSeek** = -1

class **SparseEventBlockMatcher**

Public Functions

inline explicit **SparseEventBlockMatcher**(const cv::Size &resolution, const cv::Size &windowSize = cv::Size(24, 24), const int32_t maxDisparity = 40, const int32_t minDisparity = 0, const float minScore = 1.0f)

Initialize sparse event block matcher. This constructor initializes the matcher in non-rectified space, so for accurate results the event coordinates should be already rectified.

Parameters

- **resolution** – Resolution of camera sensors. Assumes same resolution for left and right camera.

- **windowSize** – Matching window size.
- **maxDisparity** – Maximum disparity value.
- **minDisparity** – Minimum disparity value.
- **minScore** – Minimum matching score to consider matching valid.

inline explicit **SparseEventBlockMatcher**(*std*::unique_ptr<*dv*::*camera*::*StereoGeometry*> geometry, const cv::Size &windowSize = cv::Size(24, 24), const int32_t maxDisparity = 40, const int32_t minDisparity = 0, const float minScore = 1.0f)

Initialize a sparse stereo block matcher with a calibrated stereo geometry. This allows event rectification while calculating the disparity.

Parameters

- **geometry** – Stereo camera geometry.
- **windowSize** – Matching window size.
- **maxDisparity** – Maximum disparity value.
- **minDisparity** – Minimum disparity value.
- **minScore** – Minimum matching score to consider matching valid.

template<*dv*::*concepts*::*Coordinate2DIterable* **InputPoints**>
inline *std*::vector<*PixelDisparity*> **computeDisparitySparse**(const *dv*::*EventStore* &left, const *dv*::*EventStore* &right, const *InputPoints* &interestPoints)

Compute sparse disparity on given interest points. The events are accumulated sparsely only on the selected interest point regions. Returns a list of coordinates with their according disparity values, correlations and scores for each disparity match. If rectification is enabled, the returned disparity result will have `valid` flag set to false if the interest point coordinate lies outside of valid rectified pixel space.

Input event has to be passed in synchronized batches, no time validation is performed during accumulation.

Parameters

- **left** – Synchronised event batch from left camera.
- **right** – Synchronised event batch from right camera.
- **interestPoints** – List of interest coordinates in unrectified pixel space.

Returns
A list of disparity results for each input interest point.

inline const cv::Mat &**getLeftMask**() const

Get the left camera image mask. The algorithm only accumulates the frames where actual matching is going to happen. The mask will contain non-zero pixel values where accumulation needs to happen.

Returns
Interest region mask for left camera.

inline const cv::Mat &**getRightMask**() const

Get the right camera image mask. The algorithm only accumulates the frames where actual matching is going to happen. The mask will contain non-zero pixel values where accumulation needs to happen.

Returns
Interest region mask for right camera.

inline *dv*::*Frame* **getLeftFrame**() const

Get the latest accumulated left frame.

Returns
Accumulated image of the left camera from last disparity computation step.

inline *dv*::*Frame* **getRightFrame**() const

Get the latest accumulated right frame.

Returns
Accumulated image of the right camera from last disparity computation step.

inline const cv::Size &**getWindowSize**() const

Get matching window size.

Returns
Currently configured matching window size.

inline void **setWindowSize**(const cv::Size &windowSize)

Set matching window size. This is the size of cropped template image that is matched along the epipolar line of the stereo geometry.

Parameters
windowSize – New matching window size.

inline int32_t **getMaxDisparity**() const

Get maximum disparity value.

Returns
Currently configured maximum disparity value.

inline void **setMaxDisparity**(const int32_t maxDisparity)

Set maximum measured disparity. This parameter limits the matching space in pixels on the right camera image.

Parameters
maxDisparity – New maximum disparity value.

inline int32_t **getMinDisparity**() const

Get minimum disparity value.

Returns
Currently configured minimum disparity value.

inline void **setMinDisparity**(const int32_t minDisparity)

Set minimum measured disparity. This parameter limits the matching space in pixels on the right camera image.

Parameters
minDisparity – New minimum disparity value.

inline float **getMinScore**() const

Get minimum matching score value.

Returns
Currently configured minimum matching score value.

inline void **setMinScore**(const float minimumScore)

Set minimum matching score value to consider the matching valid. If matching score is below this threshold, the value for a point will be set to an invalid value and `valid` boolean to false.

Score is calculated by applying softmax function on the discrete distribution of correlation values from matching the template left patch on the epipolar line of the right camera image. This retrieves the probability mass function of the correlations. The best match is found by finding the max probability value and score is calculated for the best match by computing z-score over the probabilities.

Parameters
minimumScore – New minimum score value.

Protected Functions

template<*dv*::*concepts*::*Coordinate2D* **InputPoint**>
inline cv::Rect **getPointRoi**(const *InputPoint* &center, const int32_t offsetX, const int32_t stretchX) const

inline void **initializeMaskPoint**(cv::Mat &mask, const int32_t offsetX, const int32_t stretchX, const cv::Point2i &coord, const *std*::optional<*dv*::*camera*::*StereoGeometry*::*CameraPosition*> cameraPosition = *std*::nullopt) const

Protected Attributes

cv::Mat **mLeftMask**

cv::Mat **mRightMask**

dv::*Frame* **mLeftFrame**

dv::*Frame* **mRightFrame**

dv::*EdgeMapAccumulator* **mLeftAcc**

dv::*EdgeMapAccumulator* **mRightAcc**

cv::Size **mWindowSize**

cv::Size **mHalfWindowSize**

int32_t **mMaxDisparity**

int32_t **mMinDisparity**

float **mMinScore**

std::unique_ptr<*dv*::*camera*::*StereoGeometry*> **mStereoGeometry** = nullptr

template<class **EventStoreType**, uint32_t **patchDiameter** = 8, typename **ScalarType** = uint8_t>

class **SpeedInvariantTimeSurfaceBase** : public *dv*::*TimeSurfaceBase*<*EventStoreType*, uint8_t>

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> A speed invariant time surface, as described by https://arxiv.org/abs/1903.11332

Template Parameters

- **EventStoreType** – Type of underlying event store
- **patchDiameter** – Diameter of the patch to apply the speed invariant update. The paper defines parameter `r` which is half of the diameter value, so for an `r = 5`, use diameter = `2 * r` or 10 in this case. The update is performed using eigen optimized routines, so the value has limits: it has to be in range (0; 16) and divisible by 2. By default set to 8 which gives the best performance.

Public Functions

inline explicit **SpeedInvariantTimeSurfaceBase**(const cv::Size &shape)

Create a speed invariant time surface with known image dimensions.

Parameters
shape – Dimensions of the expected event data.

inline virtual *SpeedInvariantTimeSurfaceBase* &**operator<<**(const *EventStoreType* &store) override

Inserts the event store into the speed invariant time surface.

Parameters
store – The event store to be added

Returns
A reference to this TimeSurface.

inline virtual *SpeedInvariantTimeSurfaceBase* &**operator<<**(const typename *EventStoreType*::iterator::value_type &event) override

Inserts the event into the speed invariant time surface.

Parameters
event – The event to be added

Returns
A reference to this TimeSurface.

inline virtual void **accept**(const *EventStoreType* &store) override

Inserts the event store into the speed invariant time surface.

Parameters
store – The event store to be added

inline virtual void **accept**(const typename *EventStoreType*::iterator::value_type &event) override

Inserts the event into the speed invariant time surface.

Parameters
event – The event to be added

Protected Types

using **BaseClassType** = *TimeSurfaceBase*<*EventStoreType*, *ScalarType*>

Private Members

int64_t **mLatestPixelValue**

template<typename>

struct **std_function_exact**

std::function substitute with exact signature matching. Requires boost::callable_traits installed, which is only available with boost >= 1.66.

template<typename **R**, typename ...**Args**>

struct **std_function_exact**<*R*(*Args*...)> : public *std*::function<*R*(*Args*...)>

Public Functions

template<typename **T**, *std*::enable_if_t<*std*::is_same_v<boost::callable_traits::return_type_t<*T*>, *R*> && *std*::is_same_v<boost::callable_traits::args_t<*T*>, *std*::tuple<*Args*...>>, bool> = true>
inline **std_function_exact**(*T* &&t)

struct **StereoCalibration**

Public Functions

StereoCalibration() = default

inline **StereoCalibration**(const *std*::string &leftName, const *std*::string &rightName, const *std*::vector<float> &fundamentalMatrix_, const *std*::vector<float> &essentialMatrix_, const *std*::optional<*Metadata*> &metadata_)

inline explicit **StereoCalibration**(const pt::ptree &tree)

inline pt::ptree **toPropertyTree**() const

inline bool **operator==**(const *StereoCalibration* &rhs) const

inline Eigen::Matrix3f **getFundamentalMatrix**() const

Retrieve the fundamental matrix as Eigen::Matrix3f.

Returns
Fundamental matrix.

inline Eigen::Matrix3f **getEssentialMatrix**() const

Retrieve the essential matrix as Eigen::Matrix3f.

Returns
Essential matrix.

Public Members

std::string **leftCameraName**

Name of the left camera.

std::string **rightCameraName**

Name of the right camera.

std::vector<float> **fundamentalMatrix**

Stereo calibration Fundamental Matrix.

std::vector<float> **essentialMatrix**

Stereo calibration Essential Matrix.

std::optional<*Metadata*> **metadata**

Metadata.

class **StereoCameraRecording**

Public Functions

inline **StereoCameraRecording**(const fs::path &aedat4Path, const *std*::string &leftCameraName, const *std*::string &rightCameraName)

Create a reader for stereo camera recording. Expects at least one stream from two cameras available. Prior knowledge of stereo setup is required, otherwise it is not possible to differentiate between left and right cameras. This is just a convenience class that gives access to distinguished data streams in the recording.

Parameters

- **aedat4Path** – Path to the aedat4 file.
- **leftCameraName** – Name of the left camera.
- **rightCameraName** – Name of the right camera.

inline *MonoCameraRecording* &**getLeftReader**()

Access the left camera.

Returns
A reference to the left camera reader.

inline *MonoCameraRecording* &**getRightReader**()

Access the right camera.

Returns
A reference to the right camera reader.

Private Members

std::shared_ptr<*ReadOnlyFile*> **mReader** = nullptr

MonoCameraRecording **mLeftCamera**

MonoCameraRecording **mRightCamera**

class **StereoCameraWriter**

Public Functions

inline **StereoCameraWriter**(const fs::path &aedat4Path, const *MonoCameraWriter*::*Config* &leftConfig, const *MonoCameraWriter*::*Config* &rightConfig, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Open a file pass left / right camera configuration manually.

Parameters

- **aedat4Path** – Path to output file.
- **leftConfig** – Left camera output stream configuration.
- **rightConfig** – Right camera output stream configuration.
- **resolver** – Type resolver for the output file.

inline **StereoCameraWriter**(const fs::path &aedat4Path, const *StereoCapture* &capture, const *CompressionType* compression = *CompressionType*::*LZ4*, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Open a file and use capture device to inspect the capabilities of the cameras. This will create all possible output streams the devices can supply.

Parameters

- **aedat4Path** – Path to output file.
- **capture** – Capture object to inspect capabilities of the cameras.
- **compression** – Compression to be used for the output file.
- **resolver** – Type resolver for the output file.

Public Members

MonoCameraWriter **left**

Left writing instance.

MonoCameraWriter **right**

Right writing instance.

Private Functions

inline *std*::string **createStereoHeader**(const *dv*::*io*::*support*::*TypeResolver* &resolver)

inline void **configureStreamIds**()

Private Members

MonoCameraWriter::*Config* **leftUpdatedConfig**

MonoCameraWriter::*Config* **rightUpdatedConfig**

StreamIdContainer **leftIds**

StreamIdContainer **rightIds**

MonoCameraWriter::*StreamDescriptorMap* **mLeftOutputStreamDescriptors**

MonoCameraWriter::*StreamDescriptorMap* **mRightOutputStreamDescriptors**

std::shared_ptr<*WriteOnlyFile*> **file**

Private Static Functions

static inline void **configureCameraOutput**(int32_t &index, *dv*::*io*::*support*::*XMLTreeNode* &mRoot, *MonoCameraWriter*::*Config* &config, const *std*::string &compression, *StreamIdContainer* &ids, *MonoCameraWriter*::*StreamDescriptorMap* &streamDescriptors, const *dv*::*io*::*support*::*TypeResolver* &resolver, const *std*::string &outputPrefix)

class **StereoCapture**

Public Functions

inline **StereoCapture**(const *std*::string &leftName, const *std*::string &rightName, const *dv*::*Duration* &synchronizationTimeout = `dv::Duration(1 '000 '000)`)

Open a stereo camera setup consisting of two cameras. Finds the devices connected to the system and performs timestamp synchronization on them.

Parameters

- **leftName** – Left camera name.
- **rightName** – Right camera name.
- **synchronizationTimeout** – Timeout duration for synchronization

Throws

- `RuntimeError` – Exception if both cameras are master (missing sync cable between cameras is the most likely reason).
- `RuntimeError` – Exception is thrown if cameras fails to synchronize within given timeout duration.

Public Members

CameraCapture **left**

CameraCapture **right**

Private Static Functions

static inline void **synchronizeStereo**(*CameraCapture* &master, *CameraCapture* &secondary, const int64_t timeout)

Performs synchronization of the stereo camera setup.

Parameters

- **master** – Camera capture instance that has generates synchronization signal.
- **secondary** – Camera capture instance that receives synchronization signal.
- **timeout** – An exception is thrown if synchronization doesn't complete within given time period in microseconds.

class **StereoGeometry**

#include </builds/inivation/dv/dv-processing/include/dv-processing/camera/stereo_geometry.hpp> A class that performs stereo geometry operations and rectification of a stereo camera.

Public Types

enum class **CameraPosition**

Position enum for a single camera in a stereo configuration.

Values:

enumerator **Left**

enumerator **Right**

enum class **FunctionImplementation**

Values:

enumerator **LUT**

enumerator **SubPixel**

using **UniquePtr** = *std*::unique_ptr<*StereoGeometry*>

using **SharedPtr** = *std*::shared_ptr<*StereoGeometry*>

Public Functions

inline **StereoGeometry**(const *CameraGeometry* &leftCamera, const *CameraGeometry* &rightCamera, const *std*::vector<float> &transformToLeft, *std*::optional<cv::Size> rectifiedResolution = *std*::nullopt)

Initialize a stereo geometry class using two camera geometries for each of the stereo camera pair and a transformation matrix that describes the transformation from right camera to the left.

Parameters

- **leftCamera** – Left camera geometry.
- **rightCamera** – Right camera geometry.
- **transformToLeft** – A vector containing a homogenous transformation from right to the left camera. Vector should contain exactly 16 numbers (as per 4x4 homogenous transformation matrix) in a row-major ordering.
- **rectifiedResolution** – Resolution of the rectified image plane when remapping events/points/images from either the left or right camera (see *remapEvents()*/remapImage()). This can be the same, smaller, or larger than either the left or right camera resolutions, where downsampling/upsampling occurs if the #rectifiedResolution is smaller/larger than the camera resolution. Defaults to the left camera resolution if not provided.

inline **StereoGeometry**(const *calibrations*::*CameraCalibration* &leftCalibration, const *calibrations*::*CameraCalibration* &rightCalibration, *std*::optional<cv::Size> rectifiedResolution = *std*::nullopt)

Create a stereo geometry class from left and right camera calibration instances.

Parameters

- **leftCalibration** – Left camera calibration.
- **rightCalibration** – Right camera calibration.
- **rectifiedResolution** – Resolution of the rectified image plane when remapping events/points/images from either the left or right camera (see above constructor).

inline cv::Mat **remapImage**(const *CameraPosition* cameraPosition, const cv::Mat &image) const

Apply remapping to an input image to rectify it.

Parameters

- **cameraPosition** – Indication whether image is from left or right camera.
- **image** – Input image.

Returns

Rectified image.

inline *dv*::*EventStore* **remapEvents**(const *CameraPosition* cameraPosition, const *dv*::*EventStore* &events) const

Apply remapping on input events.

Parameters

- **cameraPosition** – Indication whether image is from left or right camera.
- **events** – Input events.

Returns
Event with rectified coordinates.

template<*dv*::*concepts*::*Coordinate2DCostructible* **OutputPoint** = cv::Point2i, *dv*::*concepts*::*Coordinate2D* **InputPoint**>
inline *std*::optional<*OutputPoint*> **remapPoint**(const *CameraPosition* cameraPosition, const *InputPoint* &point) const

Remap a point coordinate from original camera pixel space into undistorted and rectified pixel space.

Parameters

- **cameraPosition** – Camera position in the stereo setup.
- **point** – Coordinates in original camera pixel space.

Template Parameters
Point –

Returns
Undistorted and rectified coordinates or `std::nullopt` if the resulting coordinates are outside of valid output pixel range.

template<*dv*::*concepts*::*Coordinate2DCostructible* **OutputPoint** = cv::Point2i, *FunctionImplementation* **Implementation** = *FunctionImplementation*::*LUT*, *dv*::*concepts*::*Coordinate2D* **InputPoint**>
inline *OutputPoint* **unmapPoint**(const *CameraPosition* position, const *InputPoint* &point) const

Unmap a point coordinate from undistorted and rectified pixel space into original distorted pixel.

Parameters

- **position** – Camera position in the stereo setup
- **point** – Coordinates in undistorted rectified pixel space.

Template Parameters

- **OutputPoint** – Output point class
- **Implementation** – Implementation type: LUT - performs a look-up operation on a pre-computed look-up table, SubPixel - performs full computations and retrieves exact coordinates.
- **InputPoint** – Input point class (automatically inferred)

Returns
Coordinates of the pixel in original pixel space.

inline *dv*::*camera*::*CameraGeometry* **getLeftCameraGeometry**() const

Retrieve left camera geometry class that can project coordinates into stereo rectified space.

Returns
Camera geometry instance.

inline *dv*::*camera*::*CameraGeometry* **getRightCameraGeometry**() const

Retrieve right camera geometry class that can project coordinates into stereo rectified space.

Returns
Camera geometry instance.

inline *dv*::*DepthEventStore* **estimateDepth**(const cv::Mat &disparity, const *dv*::*EventStore* &events, const float disparityScale = 16.f) const

Estimate depth given the disparity map and a list of events. The coordinates will be rectified and a disparity value will be looked up in the disparity map. The depth of each event is calculated using an equation: depth = (focalLength * baseline) / disparity. Focal length is expressed in meter distance.

For practical applications, depth estimation should be evaluated prior to any use. The directly estimated depth values can contain measurable errors which should be accounted for - the errors can usually be within 10-20% fixed absolute error distance. Usually this comes from various inaccuracies and can be mitigated by introducing a correction factor for the depth estimate.

Parameters

- **disparity** – Disparity map.
- **events** – Input events.
- **disparityScale** – Scale of disparity value in the disparity map, if subpixel accuracy is enabled in the block matching, this value will be equal to 16.

Returns

A depth event store, the events will contain the same information as in the input, but additionally will have the depth value. Events whose coordinates are outside of image bounds after rectification will be skipped.

inline *dv*::*DepthFrame* **toDepthFrame**(const cv::Mat &disparity, const float disparityScale = 16.f) const

Convert a disparity map into a depth frame. Each disparity value is converted into depth using the equation depth = (focalLength * baseline) / disparity. Output frame contains distance values expressed in integer values of millimeter distance.

NOTE: Output depth frame will not have a timestamp value, it is up to the user of this method to set correct timestamp of the disparity map.

Parameters

- **disparity** – Input disparity map.
- **disparityScale** – Scale of disparity value in the disparity map, if subpixel accuracy is enabled in the block matching, this value will be equal to 16.

Returns

A converted depth frame.

Public Static Functions

static inline *std*::vector<float> **computeTransformBetween**(const *calibrations*::*CameraCalibration* &src, const *calibrations*::*CameraCalibration* &target)

Compute the homogeneous transformation that transforms a point from a source camera to a target camera based on their respective calibrations.

Parameters

- **src** – Camera calibration for the source camera.
- **target** – Camera calibration for the target camera.

Returns

4x4 transformation from source to target.

Private Functions

inline void **createLUTs**(const cv::Size &resolution, const cv::Matx33f &cameraMatrix, const cv::Mat &distortion, const cv::Mat &R, const cv::Mat &P, *std*::vector<uint8_t> &outputMask, *std*::vector<cv::Point2i> &outputRemapLUT) const

template<*concepts*::*Coordinate3DCostructible* **Output**, *concepts*::*Coordinate2D* **Input**>
inline *Output* **backProject**(const *StereoGeometry*::*CameraPosition* position, const *Input* &pixel) const

Private Members

cv::Mat **mLeftRemap1**

cv::Mat **mLeftRemap2**

cv::Mat **mRightRemap1**

cv::Mat **mRightRemap2**

cv::Mat **mLeftProjection**

cv::Mat **mRightProjection**

std::vector<uint8_t> **mLeftValidMask**

std::vector<uint8_t> **mRightValidMask**

std::vector<cv::Point2i> **mLeftRemapLUT**

std::vector<cv::Point2i> **mRightRemapLUT**

std::vector<cv::Point2i> **mLeftUnmapLUT**

std::vector<cv::Point2i> **mRightUnmapLUT**

cv::Size **mLeftResolution**

cv::Size **mRightResolution**

std::vector<float> **mDistLeft**

DistortionModel **mLeftDistModel**

std::vector<float> **mDistRight**

DistortionModel **mRightDistModel**

cv::Mat **RN**[2]

cv::Mat **Q**

dv::*kinematics*::*Transformationf* **mLeftRectifierInverse**

dv::*kinematics*::*Transformationf* **mRightRectifierInverse**

const *dv*::*camera*::*CameraGeometry* **mOriginalLeft**

const *dv*::*camera*::*CameraGeometry* **mOriginalRight**

float **mBaseline**

Private Static Functions

template<*dv*::*concepts*::*Coordinate2DCostructible* **PointType** = cv::Point2f>
static inline *std*::vector<*PointType*> **initCoordinateList**(const cv::Size &resolution)

static inline *dv*::*EventStore* **remapEventsInternal**(const *dv*::*EventStore* &events, const cv::Size &resolution, const *std*::vector<uint8_t> &mask, const *std*::vector<cv::Point2i> &remapLUT)

struct **Stream**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/stream.hpp> Structure defining a stream of data. This class holds metadata information of a stream - id, name, source, resolution (if applicable), as well as data type, compression, and other technical information needed for application to be able send or receive streams of data.

Public Functions

Stream() = default

Default constructor with no information about the stream.

inline **Stream**(const int32_t id, const *std*::string_view name, const *std*::string_view sourceName, const *std*::string_view typeIdentifier, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Manual stream configuration.

Parameters

- **id** – *Stream* ID.
- **name** – Name of the stream.

- **sourceName** – *Stream* source, usually a camera name.
- **typeIdentifier** – Flatbuffer compiler generated type identifier string, unique for the stream type.
- **resolver** – Type resolver, supports default streams, used only for custom generated type support.

inline void **addMetadata**(const *std*::string &name, const *dv*::*io*::*support*::*VariantValueOwning* &value)

Add metadata to the stream. If an entry already exists, it will be replaced with the new value.

Parameters

- **name** – Name of the metadata entry.
- **value** – Metadata value.

inline *std*::optional<*dv*::*io*::*support*::*VariantValueOwning*> **getMetadataValue**(const *std*::string_view name) const

Get a metadata attribute value.

Parameters

name – Name of a metadata attribute.

Returns

Metadata value in a variant or std::nullopt if it's not found.

inline void **setTypeDescription**(const *std*::string &description)

Set type description. This only sets type description metadata field.

Parameters

description – Metadata string that describes the type in this stream.

inline void **setModuleName**(const *std*::string &moduleName)

Set module name that originally produces the data. This only sets the original module name metadata field.

Parameters

moduleName – Module name that originally produces the data.

inline void **setOutputName**(const *std*::string &outputName)

Set original output name. This only sets the original output name metadata field.

Parameters

outputName – Name of the output that produces the data, usually referring to DV module output.

inline void **setCompression**(const *dv*::*CompressionType* compression)

Set compression metadata field for this stream. This only sets the metadata field of this stream.

Parameters

compression – Type of compression.

inline *std*::optional<*std*::string> **getTypeDescription**() const

Get type description.

Returns

Type description string if available, std::nullopt otherwise.

inline *std*::optional<*std*::string> **getModuleName**() const

Get module name.

Returns

Module name string if available, std::nullopt otherwise.

inline *std*::optional<*std*::string> **getOutputName**() const

Get output name.

Returns
Output name string if available, std::nullopt otherwise.

inline *std*::optional<*dv*::*CompressionType*> **getCompression**() const

Get compression type string.

Returns
compression type string if available, std::nullopt otherwise.

inline void **setAttribute**(const *std*::string_view name, const *dv*::*io*::*support*::*VariantValueOwning* &value)

Set an attribute of this stream, if the attribute field does not exist, it will be created.

Parameters

- **name** – Name of the attribute.
- **value** – Attribute value.

inline *std*::optional<*dv*::*io*::*support*::*VariantValueOwning*> **getAttribute**(const *std*::string_view name) const

Get attribute value given it's name.

Parameters
name – Name of the attribute.

Returns
Return variant of the value if the an attribute with given name exists, std::nullopt otherwise.

template<typename **Type**>
inline *std*::optional<*Type*> **getAttributeValue**(const *std*::string_view name) const

Get attribute value given it's name.

Template Parameters
Type – Type of the attribute.

Parameters
name – Name of the attribute.

Returns
Return the attribute value if the an attribute with given name exists, std::nullopt otherwise.

inline *std*::optional<cv::Size> **getResolution**() const

Get resolution of this stream by parsing metadata.

Returns
Stream resolution or `std::nullopt` if resolution is not available.

inline void **setResolution**(const cv::Size &resolution)

Set the stream resolution in the metadata of this stream.

Parameters
resolution – *Stream* resolution.

inline *std*::optional<*std*::string> **getSource**() const

Get source name (usually the camera name) from metadata of the stream.

Returns
Stream source or `std::nullopt` if a source name is not available.

inline void **setSource**(const *std*::string &source)

Set a source name of this stream, usually camera name.

Parameters

source – Source name, usually camera name string.

Public Members

int32_t **mId** = 0

Stream ID.

std::string **mName**

Name of the stream.

std::string **mTypeIdentifier**

Stream type identifier.

dv::*types*::*Type* **mType**

Internal type definition.

dv::*io*::*support*::*XMLTreeNode* **mXMLNode**

XML tree node that can be used to encode information about the stream.

Public Static Functions

static inline *Stream* **EventStream**(const int32_t id, const *std*::string &name, const *std*::string &sourceName, const cv::Size &resolution)

Create an event stream.

Parameters

- **id** – *Stream* ID.
- **name** – Name of the stream.
- **sourceName** – *Stream* source, usually a camera name.
- **resolution** – Event sensor resolution.

Returns

Stream definition.

static inline *Stream* **FrameStream**(const int32_t id, const *std*::string &name, const *std*::string &sourceName, const cv::Size &resolution)

Create a frame stream.

Parameters

- **id** – *Stream* ID.
- **name** – Name of the stream.
- **sourceName** – *Stream* source, usually a camera name.
- **resolution** – *Frame* sensor resolution.

Returns
Stream definition.

static inline *Stream* **IMUStream**(const int32_t id, const *std*::string &name, const *std*::string &sourceName)

Create an *IMU* stream.

Parameters

- **id** – *Stream* ID.
- **name** – Name of the stream.
- **sourceName** – *Stream* source, usually a camera name.

Returns
Stream definition.

static inline *Stream* **TriggerStream**(const int32_t id, const *std*::string &name, const *std*::string &sourceName)

Create an triger stream.

Parameters

- **id** – *Stream* ID.
- **name** – Name of the stream.
- **sourceName** – *Stream* source, usually a camera name.

Returns
Stream definition.

template<class **Type**>
static inline *Stream* **TypedStream**(const int32_t id, const *std*::string &name, const *std*::string &sourceName, const *dv*::*io*::*support*::*TypeResolver* &resolver = *dv*::*io*::*support*::*defaultTypeResolver*)

Create a stream by providing providing a stream type packet type as a template parameter.

Template Parameters
Type – Type of the stream.

Parameters

- **id** – *Stream* ID.
- **name** – Name of the stream.
- **sourceName** – *Stream* source, usually a camera name.
- **resolver** – Type resolver, supports default streams, used only for custom generated type support.

Returns
Stream definition.

struct **StreamDescriptor**

Public Functions

inline explicit **StreamDescriptor**(const *Stream* &stream)

Public Members

size_t **mSeekIndex** = 0

dv::*io*::*Stream* **mStream**

std::map<*std*::string, *std*::string> **mMetadata**

struct **StreamDescriptor**

Public Functions

inline **~StreamDescriptor**()

inline **StreamDescriptor**(uint32_t id, const *types*::*Type* *type)

Public Members

uint32_t **id**

const *dv*::*types*::*Type* ***type**

int64_t **lastTimestamp**

void ***elementBuffer**

std::function<void(void*)> **freeElementBufferCall** = nullptr

struct **StreamIdContainer**

Public Members

int32_t **mEventStreamId** = -1

int32_t **mImuStreamId** = -1

int32_t **mTriggerStreamId** = -1

int32_t **mFrameStreamId** = -1

template<class **PacketType**>

class **StreamSlicer**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/stream_slicer.hpp> The *StreamSlicer* is a class that takes on incoming timestamped data, stores them in a minimal way and invokes functions at individual periods.

Public Functions

StreamSlicer() = default

inline void **accept**(const *PacketType* &data)

Add a full packet to the streaming buffer and evaluate jobs. This function copies the data over.

Parameters
data – the packet to be added to the buffer.

template<class **ElementType**>
inline void **accept**(const *ElementType* &element)

Adds a single element of a stream to the slicer buffer and evaluate jobs.

Parameters
element – the element to be added to the buffer

inline void **accept**(*PacketType* &&packet)

Adds full stream packet of data to the buffer and evaluates jobs.

Parameters
packet – the packet to be added to the buffer

inline int **doEveryNumberOfEvents**(const size_t n, *std*::function<void(*PacketType*&)> callback)

Adds a number-of-elements triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback function every `n` elements are added to the stream buffer, with the corresponding data. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Deprecated:

Use *doEveryNumberOfElements()* method instead.

Parameters

- **n** – the interval (in number of elements) in which the callback should be called
- **callback** – the callback function that gets called on the data every interval

Returns
A handle to uniquely identify the job.

inline int **doEveryNumberOfElements**(const size_t n, *std*::function<void(*PacketType*&)> callback)

Adds a number-of-elements triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback function every `n` elements are added to the stream buffer, with the corresponding data. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Parameters

- **n** – the interval (in number of elements) in which the callback should be called
- **callback** – the callback function that gets called on the data every interval

Returns
A handle to uniquely identify the job.

inline int **doEveryNumberOfElements**(const size_t n, *std*::function<void(const *dv*::*TimeWindow*&, *PacketType*&)> callback)

Adds a number-of-elements triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback function every `n` elements are added to the stream buffer, with the corresponding data. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Parameters

- **n** – the interval (in number of elements) in which the callback should be called
- **callback** – the callback function that gets called on the data every interval, also passes time window containing the inter

Returns
A handle to uniquely identify the job.

inline int **doEveryTimeInterval**(const *dv*::*Duration* interval, *std*::function<void(const *PacketType*&)> callback)

Adds an element-timestamp-interval triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback whenever the timestamp difference of an incoming event to the last time the function was called is bigger than the interval. As the timing is based on event times rather than CPU time, the actual time periods are not guaranteed, especially with a low event rate. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Parameters

- **interval** – the interval in which the callback should be called
- **callback** – the callback function that gets called on the data every interval

Returns
A handle to uniquely identify the job.

inline int **doEveryTimeInterval**(const int64_t microseconds, *std*::function<void(const *PacketType*&)> callback)

Adds an element-timestamp-interval triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback whenever the timestamp difference of an incoming event to the last time the function was called is bigger than the interval. As the timing is based on event times rather than CPU time, the actual time periods are not guaranteed, especially with a low event rate. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Deprecated:

Please pass interval parameter using dv::Duration.

Parameters

- **interval** – the interval in which the callback should be called
- **callback** – the callback function that gets called on the data every interval

Returns
A handle to uniquely identify the job.

inline int **doEveryTimeInterval**(const *dv*::*Duration* interval, *std*::function<void(const *dv*::*TimeWindow*&, const *PacketType*&)> callback)

Adds an element-timestamp-interval triggered job to the Slicer. A job is defined by its interval and callback function. The slicer calls the callback whenever the timestamp difference of an incoming event to the last time the function was called is bigger than the interval. As the timing is based on event times rather than CPU time, the actual time periods are not guaranteed, especially with a low event rate. The (cpu) time interval between individual calls to the function depends on the physical event rate as well as the bulk sizes of the incoming data.

Parameters

- **interval** – the interval in which the callback should be called
- **callback** – the callback function that gets called with the time window information and the data as arguments every interval

Returns
An id to uniquely identify the job.

inline bool **hasJob**(const int jobId) const

Returns true if the slicer contains the slicejob with the provided id

Parameters
jobId – the id of the slicejob in question

Returns
true, if the slicer contains the given slicejob

inline void **removeJob**(const int jobId)

Removes the given job from the list of current jobs.

Parameters
jobId – The job id to be removed

inline void **modifyTimeInterval**(const int jobId, const int64_t timeInterval)

Modifies the time interval of the supplied job to the requested value

Deprecated:

Please pass time interval as dv::Duration instead.

Parameters

- **jobId** – the job whose time interval should be changed
- **timeInterval** – the new time interval value

inline void **modifyTimeInterval**(const int jobId, const *dv*::*Duration* timeInterval)

Modifies the time interval of the supplied job to the requested value

Parameters

- **jobId** – the job whose time interval should be changed
- **timeInterval** – the new time interval value

inline void **modifyNumberInterval**(const int jobId, const size_t numberInterval)

Modifies the number interval of the supplied job to the requested value

Parameters

- **jobId** – the job whose number interval should be changed
- **numberInterval** – the new number interval value

Private Functions

inline void **evaluate**()

Should get called as soon as there is fresh data available. It loops through all jobs and determines if they can run on the new data. The jobs get executed as often as possible. Afterwards, all data that has been processed by all jobs gets discarded.

Private Members

PacketType **mStorePacket**

Global storage packet that holds just as many data elements as minimally required for all outstanding calls.

std::map<int, SliceJob> **mSliceJobs**

List of all the sliceJobs.

int **mHashCounter** = 0

class **TCPTLSSocket** : public *dv*::*io*::*network*::*SocketBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/network/tcp_tls_socket.hpp> Minimal wrapper of TCP socket with optional TLS encryption.

Public Functions

inline **TCPTLSSocket**(*asioTCP*::socket &&socket, const bool tlsEnabled, const asioSSL::stream_base::handshake_type tlsHandshake, asioSSL::context &tlsContext)

Create a TCP socket with optional TLS encryption.

Parameters

- **socket** – A connected TCP socket instance.
- **tlsEnabled** – Whether TLS encryption is enabled, if true, TLS handshake will be immediately performed during construction.
- **tlsHandshake** – Type of TLS handshake, this is ignored if TLS is disabled.
- **tlsContext** – Pre-configured TLS context for encryption.

inline **~TCPTLSSocket**() override

inline virtual bool **isOpen**() const override

Check whether socket is open and active.

Returns
True if socket is open, false otherwise.

inline bool **isSecured**() const

Check whether socket has encryption enabled.

Returns
True if socket has encryption enabled, false otherwise.

inline virtual void **close**() override

Close underlying TCP socket cleanly.

inline virtual void **write**(const asio::const_buffer &buf, *SocketBase*::*CompletionHandler* &&wrHandler) override

Write handler needs following signature: void (const boost::system::error_code &, size_t)

inline virtual void **read**(const asio::mutable_buffer &buf, *SocketBase*::*CompletionHandler* &&rdHandler) override

Read handler needs following signature: void (const boost::system::error_code &, size_t)

inline virtual void **syncWrite**(const asio::const_buffer &buf) override

Blocking write data to the socket.

Parameters
buf – Data to write.

inline virtual void **syncRead**(const asio::mutable_buffer &buf) override

Blocking read from socket.

Parameters
buf – Buffer for data to be read into.

inline *asioTCP*::endpoint **local_endpoint**() const

Retrieve local endpoint.

Returns
Local endpoint.

inline asioIP::address **local_address**() const

Get the local address of the current endpoint.

Returns
IP address of the local connection.

inline uint16_t **local_port**() const

Get local port number.

Returns
Local port number.

inline *asioTCP*::endpoint **remote_endpoint**() const

inline asioIP::address **remote_address**() const

Remote endpoint IP address.

Returns
Remote endpoint IP address.

inline uint16_t **remote_port**() const

Get remote endpoint port number.

Returns

Remote endpoint port number.

Private Functions

inline *asioTCP*::socket &**baseSocket**()

Private Members

asioTCP::endpoint **mLocalEndpoint**

asioTCP::endpoint **mRemoteEndpoint**

asioSSL::stream<*asioTCP*::socket> **mSocket**

bool **mSocketClosed** = false

bool **mSecureConnection** = false

struct **TimedKeyPoint** : public *flatbuffers*::NativeTable

Public Types

typedef *TimedKeyPointFlatbuffer* **TableType**

Public Functions

inline **TimedKeyPoint**()

inline **TimedKeyPoint**(const Point2f &_pt, float _size, float _angle, float _response, int32_t _octave, int32_t _class_id, int64_t _timestamp)

Public Members

Point2f **pt**

float **size**

float **angle**

float **response**

int32_t **octave**

int32_t **class_id**

int64_t **timestamp**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

struct **TimedKeyPointBuilder**

Public Functions

inline void **add_pt**(const Point2f *pt)

inline void **add_size**(float size)

inline void **add_angle**(float angle)

inline void **add_response**(float response)

inline void **add_octave**(int32_t octave)

inline void **add_class_id**(int32_t class_id)

inline void **add_timestamp**(int64_t timestamp)

inline explicit **TimedKeyPointBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

TimedKeyPointBuilder &**operator=**(const *TimedKeyPointBuilder*&)

inline *flatbuffers*::Offset<*TimedKeyPointFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **TimedKeyPointFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *TimedKeyPoint* **NativeTableType**

Public Functions

inline const Point2f *__pt__() const

coordinates of the keypoints.

inline float __size__() const

diameter of the meaningful keypoint neighborhood.

inline float __angle__() const

computed orientation of the keypoint (-1 if not applicable); it's in [0,360) degrees and measured relative to image coordinate system, ie in clockwise.

inline float __response__() const

the response by which the most strong keypoints have been selected. Can be used for the further sorting or subsampling.

inline int32_t __octave__() const

octave (pyramid layer) from which the keypoint has been extracted.

inline int32_t __class_id__() const

object class (if the keypoints need to be clustered by an object they belong to).

inline int64_t __timestamp__() const

Timestamp (µs).

inline bool __Verify__(*flatbuffers*::Verifier &verifier) const

inline *TimedKeyPoint* *__UnPack__(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void __UnPackTo__(*TimedKeyPoint* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable *__MiniReflectTypeTable__()

static inline constexpr const char *__GetFullyQualifiedName__()

static inline void __UnPackToFrom__(*TimedKeyPoint* *_o, const *TimedKeyPointFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*TimedKeyPointFlatbuffer*> __Pack__(*flatbuffers*::FlatBufferBuilder &_fbb, const *TimedKeyPoint* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

struct __TimedKeyPointPacket__ : public *flatbuffers*::NativeTable

Public Types

typedef *TimedKeyPointPacketFlatbuffer* **TableType**

Public Functions

inline **TimedKeyPointPacket**()

inline **TimedKeyPointPacket**(const *dv*::*cvector*<*TimedKeyPoint*> &_elements)

Public Members

dv::*cvector*<*TimedKeyPoint*> **elements**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *TimedKeyPointPacket* &packet)

struct **TimedKeyPointPacketBuilder**

Public Functions

inline void **add_elements**(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*TimedKeyPointFlatbuffer*>>> elements)

inline explicit **TimedKeyPointPacketBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

TimedKeyPointPacketBuilder &**operator=**(const *TimedKeyPointPacketBuilder*&)

inline *flatbuffers*::Offset<*TimedKeyPointPacketFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **TimedKeyPointPacketFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *TimedKeyPointPacket* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*TimedKeyPointFlatbuffer*>> ***elements**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *TimedKeyPointPacket* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*TimedKeyPointPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*TimedKeyPointPacket* *_o, const *TimedKeyPointPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*TimedKeyPointPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *TimedKeyPointPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "TKPS"

struct **TimeElementExtractor**

Public Functions

inline constexpr **TimeElementExtractor**() noexcept

inline constexpr **TimeElementExtractor**(const int64_t startTimestamp_, const int64_t endTimestamp_) noexcept

~TimeElementExtractor() = default

TimeElementExtractor(const *TimeElementExtractor* &t) = default

TimeElementExtractor &**operator=**(const *TimeElementExtractor* &rhs) = default

TimeElementExtractor(*TimeElementExtractor* &&t) = default

TimeElementExtractor &**operator=**(*TimeElementExtractor* &&rhs) = default

inline constexpr bool **operator==**(const *TimeElementExtractor* &rhs) const noexcept

inline constexpr bool **operator!=**(const *TimeElementExtractor* &rhs) const noexcept

Public Members

int64_t **startTimestamp**

int64_t **endTimestamp**

int64_t **numElements**

template<class **EventStoreType**, typename **ScalarType** = int64_t>

class **TimeSurfaceBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/core.hpp> TimeSurface class that builds the surface of the occurrences of the last timestamps.

Subclassed by *dv::SpeedInvariantTimeSurfaceBase< EventStoreType, patchDiameter, ScalarType >*

Public Types

using **Scalar** = *ScalarType*

Public Functions

TimeSurfaceBase() = default

Dummy constructor Constructs a new, empty TimeSurface without any data allocated to it.

inline explicit **TimeSurfaceBase**(const uint32_t rows, const uint32_t cols)

Creates a new TimeSurface with the given size. The Mat is zero initialized

Parameters

- **rows** – The number of rows of the TimeSurface
- **cols** – The number of cols of the TimeSurface

inline explicit **TimeSurfaceBase**(const cv::Size &size)

Creates a new TimeSurface of the given size. The Mat is zero initialized.

Parameters

size – The opencv size to be used to initialize

TimeSurfaceBase(const *TimeSurfaceBase* &other) = default

Copy constructor, constructs a new time surface with shared ownership of the data.

Parameters

other – The time surface to be copied. The data is not copied but takes shared ownership.

virtual **~TimeSurfaceBase**() = default

Destructor

inline virtual *TimeSurfaceBase* &**operator<<**(const *EventStoreType* &store)

Inserts the event store into the time surface.

Parameters
store – The event store to be added

Returns
A reference to this *TimeSurfaceBase*.

inline virtual *TimeSurfaceBase* &**operator<<**(const typename *EventStoreType*::iterator::value_type &event)

Inserts the event into the time surface.

Parameters
event – The event to be added

Returns
A reference to this *TimeSurfaceBase*.

inline *dv*::*Frame* &**operator>>**(*dv*::*Frame* &mat) const

Generates a frame from the data contained in the event store

Parameters
mat – The storage where the frame should be generated

Returns
A reference to the generated frame.

inline virtual void **accept**(const *EventStoreType* &store)

Inserts the event store into the time surface.

Parameters
store – The event store to be added

inline virtual void **accept**(const typename *EventStoreType*::iterator::value_type &event)

Inserts the event into the time surface.

Parameters
event – The event to be added

inline const *ScalarType* &**at**(const int16_t y, const int16_t x) const

Returns a const reference to the element at the given coordinates. The element can only be read from

Parameters

- **y** – The y coordinate of the element to be accessed.
- **x** – The x coordinate of the element to be accessed.

Returns
A const reference to the element at the requested coordinates.

inline *ScalarType* &**at**(const int16_t y, const int16_t x)

Returns a reference to the element at the given coordinates. The element can both be read from as well as written to.

Parameters

- **y** – The y coordinate of the element to be accessed.
- **x** – The x coordinate of the element to be accessed.

Returns
A reference to the element at the requested coordinates.

inline const *ScalarType* &**operator()**(const int16_t y, const int16_t x) const noexcept

Returns a const reference to the element at the given coordinates. The element can only be read from

Parameters

- **y** – The y coordinate of the element to be accessed.
- **x** – The x coordinate of the element to be accessed.

Returns
A const reference to the element at the requested coordinates.

inline *ScalarType* &**operator()**(const int16_t y, const int16_t x) noexcept

Returns a reference to the element at the given coordinates. The element can both be read from as well as written to.

Parameters

- **y** – The y coordinate of the element to be accessed.
- **x** – The x coordinate of the element to be accessed.

Returns
A reference to the element at the requested coordinates.

inline auto **block**(const int16_t topRow, const int16_t leftCol, const int16_t height, const int16_t width) const

Returns a block of the time surface

Parameters

- **topRow** – the row coordinate at the top of the block
- **leftCol** – the column coordinate at the left of the block
- **height** – the height of the block
- **width** – the width of the block

Returns
the block

inline auto **block**(const int16_t topRow, const int16_t leftCol, const int16_t height, const int16_t width)

Returns a block of the time surface

Parameters

- **topRow** – the row coordinate at the top of the block
- **leftCol** – the column coordinate at the left of the block
- **height** – the height of the block
- **width** – the width of the block

Returns
the block

inline *dv*::*Frame* **generateFrame**() const

Generates a frame from the data contained in the event store

Returns
The generated frame.

template<class **T** = uint8_t>
inline *std*::pair<cv::Mat, int64_t> **getOCVMat**() const

Creates a new OpenCV matrix of the type given and copies the time data into this OpenCV matrix. This version does only subtracts an offset from the values for them to fit into the value range of the requested frame type. Therefore this method preserves the units of the timestamps that are contained in the time surface.

The data in the time surface is of signed 64bit integer type. There is no OpenCV type that can hold the full range of these values. Therefore, the returned data is a pair of an OpenCV Mat, of a type that can be chosen by the user, and an offset of signed 64bit integer, which contains the offset that can be added to each pixel value so that their values are in units of microseconds.

Template Parameters
T – The type of the OpenCV Mat to be generated.

Returns
An OpenCV Mat of the requested type, as well as an offset which can be added to the matrix in order for the data to be in microseconds.

template<typename **T** = uint8_t>
inline cv::Mat **getOCVMatScaled**(const *std*::optional<int64_t> lookBackOverride = *std*::nullopt) const

Creates a new OpenCV matrix of the type given and copies the time data into this OpenCV matrix. This version scales the values for them to fit into the value range of the requested frame type. Therefore the units of the timestamps are not preserved.

The data in the time surface is of signed 64bit integer type. There is no OpenCV type that can hold the full range of these values. Therefore, the returned data is a pair of an OpenCV Mat, of a type that can be chosen by the user, and an offset of signed 64bit integer, which contains the offset that can be added to each pixel value so that their values are in units of microseconds.

Template Parameters
T – The type of the OpenCV Mat to be generated.

Parameters
lookBackOverride – override the amount of time to look back into the past. Defaults to the complete range contained in the time surface. The unit of the parameter is the unit of time contained in the TimeSurface.

Returns
An OpenCV Mat of the requested type, as well as an offset which can be added to the matrix in order for the data to be in microseconds.

inline void **reset**()

Sets all values in the time surface to zero

template<typename **T**>
inline *TimeSurfaceBase* **operator+**(const *T* &s) const

Adds a constant to the time surface. Values are bounds checked to 0. If the new time would become negative, it is set to 0.

Template Parameters
T – The type of the constant. Accepts any numeric type.

Parameters
s – The constant to be added

Returns
A new *TimeSurfaceBase* with the changed times

template<typename **T**>

inline *TimeSurfaceBase* &**operator+=**(const *T* &s)

Adds a constant to the TimeSurface. Values are bounds checked to 0. If the new time would become negative, it is set to 0.

Template Parameters
T – The type of the constant. Accepts any numeric type.

Parameters
s – The constant to be added

Returns
A reference to the *TimeSurfaceBase*

template<typename **T**>
inline *TimeSurfaceBase* **operator-**(const *T* &s) const

Subtracts a constant from the TimeSurface. Values are bounds checked to 0. If the new time would become negative, it is set to 0.

Template Parameters
T – The type of the constant. Accepts any numeric type.

Parameters
s – The constant to be subtracted

Returns
A reference to the *TimeSurfaceBase*

template<typename **T**>
inline *TimeSurfaceBase* &**operator-=**(const *T* &s)

Subtracts a constant from the TimeSurface. Values are bounds checked to 0. If the new time would become negative, it is set to 0.

Template Parameters
T – The type of the constant. Accepts any numeric type.

Parameters
s – The constant to be subtracted

Returns
A reference to the *TimeSurfaceBase*

template<typename **T**>
inline *TimeSurfaceBase* &**operator=**(const *T* &s)

Assigns constant to the TimeSurface. Values are bounds checked to 0. If the new time would become negative, it is set to 0.

Template Parameters
T – The type of the constant. Accepts any numeric type.

Parameters
s – The constant to be subtracted

Returns
A reference to the *TimeSurfaceBase*

inline cv::Size **size**() const noexcept

The size of the TimeSurface.

Returns
Returns the size of this time matrix as an opencv size

inline int16_t **rows**() const noexcept

Returns the number of rows of the TimeSurface

Returns
the number of rows

inline int16_t **cols**() const noexcept

Returns the number of columns of the TimeSurface

Returns
the number of columns

inline bool **empty**() const noexcept

Returns true if the TimeSurface has zero size. In this case, it was not allocated with a size.

Deprecated:

Use *isEmpty()* instead.

See also:

TimeSurfaceBase::isEmpty()

Returns
true if the TimeSurface does not have a size > 0

inline bool **isEmpty**() const noexcept

Returns true if the TimeSurface has zero size. In this case, it was not allocated with a size.

Returns
true if the TimeSurface does not have a size > 0

Protected Functions

inline void **addImpl**(const *ScalarType* a, *TimeSurfaceBase* &target) const

Protected Attributes

Eigen::Matrix<*ScalarType*, Eigen::Dynamic, Eigen::Dynamic> **mData**

struct **TimeWindow**

Public Functions

inline **TimeWindow**(const int64_t timestamp, const *dv*::*Duration* duration)

inline **TimeWindow**(const int64_t startTime, const int64_t endTime)

inline *dv*::*Duration* **duration**() const

Public Members

int64_t **startTime**

int64_t **endTime**

class **TrackerBase**

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/tracker_base.hpp> A base class for implementing feature trackers, that track sets of features against streams of various inputs. This class specifically does not define an input type, so it could be defined by the specific implementation.

Subclassed by *dv::features::ImageFeatureLKTracker*, *dv::features::MeanShiftTracker*

Public Types

typedef *std*::shared_ptr<*TrackerBase*> **SharedPtr**

typedef *std*::unique_ptr<*TrackerBase*> **UniquePtr**

Public Functions

inline void **setMaxTracks**(size_t _maxTracks)

Set the maximum number of tracks.

Parameters
_maxTracks – Maximum number of tracks

inline size_t **getMaxTracks**() const

Get the maximum number of tracks.

Returns
Maximum number of tracks

inline const *Result*::*SharedPtr* &**getLastFrameResults**() const

Retrieve cached last frame detection results.

Returns
Detection result from the last processed frame.

inline *Result*::*ConstPtr* **runTracking**()

Performed the tracking and cache the results.

Returns
Tracking result.

virtual **~TrackerBase**() = default

inline virtual void **removeTracks**(const *std*::vector<int> &trackIds)

Remove tracks from cached results, so the wouldn't be tracked anymore. TrackIds are the `class_id` value of the keypoint structure.

Parameters
trackIds – Track `class_id` values to be removed from cached tracker results.

Protected Functions

virtual *Result*::*SharedPtr* **track**() = 0

Virtual function that is called after all inputs were set. This function should perform tracking against `lastFrameResults`.

Returns
Tracking result.

Protected Attributes

size_t **maxTracks** = 200

Maximum number of tracks.

Result::*SharedPtr* **lastFrameResults**

Cached results of last tracker execution.

template<*std*::floating_point **Scalar**>

class **Transformation**

#include </builds/inivation/dv/dv-processing/include/dv-processing/kinematics/transformation.hpp> Basic transformation wrapper containing homogenous 3D transformation and timestamp.

Template Parameters
Scalar – Customizable storage type - float or double.

Public Functions

inline EIGEN_MAKE_ALIGNED_OPERATOR_NEW **Transformation**(int64_t timestamp, const Eigen::Matrix<*Scalar*, 4, 4> &T)

Construct the transformation from a timestamp and 4x4 transformation matrix

Parameters

- **timestamp** – Unix timestamp in microsecond format
- **T** – Homogenous 3D transformation matrix

inline **Transformation**()

Construct an identity transformation from with timestamp.

Parameters
timestamp – Unix timestamp in microsecond format

inline **Transformation**(int64_t timestamp, const Eigen::Matrix<*Scalar*, 3, 1> &translation, const Eigen::Quaternion<*Scalar*> &rotation)

Construct the transformation from timestamp, 3D translation vector and quaternion describing the rotation.

Parameters

- **timestamp** – Unix timestamp in microsecond format
- **translation** – 3D translation vector
- **rotation** – Quaternion describing the rotation

inline **Transformation**(int64_t timestamp, const Eigen::Matrix<*Scalar*, 3, 1> &translation, const Eigen::Matrix<*Scalar*, 3, 3> &rotationMatrix)

Construct the transformation from timestamp, 3D translation vector and quaternion describing the rotation.

Parameters

- **timestamp** – Unix timestamp in microsecond format
- **translation** – 3D translation vector
- **rotationMatrix** – Rotation matrix describing the rotation

inline **Transformation**(int64_t timestamp, const cv::Mat &translation, const cv::Mat &rotation)

Construct the transformation from timestamp, 3D translation vector and quaternion describing the rotation.

Parameters

- **timestamp** – Unix timestamp in microsecond format
- **translation** – 3D translation vector
- **rotation** – 3x3 rotation matrix

inline int64_t **getTimestamp**() const

Get timestamp.

Returns
Unix timestamp of the transformation in microseconds.

inline const Eigen::Matrix<*Scalar*, 4, 4> &**getTransform**() const

Get the transformation matrix.

Returns
Transformation matrix in 4x4 format

inline Eigen::Matrix<*Scalar*, 3, 3> **getRotationMatrix**() const

Retrieve a copy of 3x3 rotation matrix.

Returns
3x3 rotation matrix

inline Eigen::Quaternion<*Scalar*> **getQuaternion**() const

Retrieve rotation expressed as a quaternion.

Returns
Quaternion containing rotation.

template<*concepts*::*Coordinate3DCostructible* **Output** = Eigen::Matrix<*Scalar*, 3, 1>>
inline *Output* **getTranslation**() const

Retrieve translation as 3D vector.

Returns
Vector containing translation.

template<*concepts*::*Coordinate3DCostructible* **Output** = Eigen::Matrix<*Scalar*, 3, 1>, *concepts*::*Coordinate3D* **Input**>
inline *Output* **transformPoint**(const *Input* &point) const

Transform a point using this transformation.

Parameters
point – Point to be transformed

Returns
Transformed point

template<*concepts*::*Coordinate3DCostructible* **Output** = Eigen::Matrix<*Scalar*, 3, 1>, *concepts*::*Coordinate3D* **Input**>
inline *Output* **rotatePoint**(const *Input* &point) const

Apply rotation only transformation on the given point.

Parameters
point – Point to be transformed

Returns
Transformed point

inline *Transformation*<*Scalar*> **inverse**() const

Calculate the inverse homogenous transformation of this transform.

Returns
Inverse transformation with the current timestamp.

inline *Transformation*<*Scalar*> **delta**(const *Transformation*<*Scalar*> &target) const

Find the transformation from current to target. (T_target_current s.t. p_target = T_target_current*p_current).

Parameters
target – Target transformation.

Returns
Transformation from this to target.

Public Static Functions

static inline *Transformation* **fromNonHomogenous**(int64_t timestamp, const Eigen::Matrix<*Scalar*, 3, 4> &T)

Construct the transformation from a timestamp and 3x4 non-homogenous transformation matrix.

Parameters

- **timestamp** – Unix timestamp in microsecond format
- **T** – 3x4 3D transformation matrix

Private Members

int64_t **mTimestamp**

Timestamp of the transformation, Unix timestamp in microseconds.

Eigen::Matrix<*Scalar*, 4, 4> **mT**

The transformation itself, stored in 4x4 format:

R|*T*

0|1

class **TranslationLossFunctor** : public *dv*::*optimization*::*OptimizationFunctor*<float>

#include </builds/inivation/dv/dv-processing/include/dv-processing/optimization/contrast_maximization_translation_and_depth.hpp>
Given a chunk of events, the idea of contrast maximization is to warp events in space and time given a predefined motion model. Contrast maximization aims at finding the optimal parameters of the given motion model. The idea is that if the motion is perfectly estimated, all events corresponding to the same point in the scene, will be warped to the same image plane location, at a given point in time. If this happens, the reconstructed event image will be sharp, having high contrast. This high contrast is measured as variance in the image. For this reason, contrast maximization searches for the best motion parameters which maximize the contrast of the event image reconstructed after warping events in space to a specific point in time. In order to warp event in space and time we use the "dv::kinematics::MotionCompensator" class. This contrast maximization class assumes pure camera translation motion model. Given a set of events in a time range (init_time, end_time), assuming a constant translational speed between init_time and end_time, translation (x, y, z) and scene depth are optimized to maximize contrast of event image. Since the speed is assumed to be constant between init_time end end_time, the camera position at time t_k is computed as : t_k = speed*dt, where dt = t_k - init_time. The scene depth is included in the optimization since it is strongly correlated to the camera translation. Scene depth is assumed to be constant between init_time and end_time.

Public Functions

inline **TranslationLossFunctor**(*dv*::*camera*::*CameraGeometry*::*SharedPtr* &camera, const *dv*::*EventStore* &events, float contribution, int inputDim, int numMeasurements)

This contrast maximization class assumes pure camera translation motion model. Given a set of events in a time range (init_time, end_time), assuming a constant translational speed between init_time and end_time, translation (x, y, z) and scene depth are optimized to maximize contrast of event image.

Parameters

- **camera** – Camera geometry used to create motion compensator
- **events** – Events used to compute motion compensated image
- **contribution** – Contribution value of each event to the total pixel intensity
- **inputDim** – Number of parameters to optimize
- **numMeasurements** – Number of function evaluation performed to compute the gradient

inline virtual int **operator()**(const Eigen::VectorXf &translationAndDepth, Eigen::VectorXf &stdInverse) const

Implementation of the objective function: optimize camera translation (x, y, z) and scene depth. Current cost is stored in stdInverse. Notice that since we want to maximize the contrast but optimizer minimize cost function we use as cost 1/contrast

Private Members

dv::*camera*::*CameraGeometry*::*SharedPtr* **mCamera**

Camera geometry data. This information is used to create motionCompensator and compensate events.

const *dv*::*EventStore* **mEvents**

Raw events compensated using translation along x, y, z and current scene depth.

const float **mContribution**

Event contribution for total pixel intensity. This parameter is very important since it strongly influence contrast value. It needs to be tuned based on scene and length of event chunk.

struct **Trigger** : public *flatbuffers*::NativeTable

Public Types

typedef *TriggerFlatbuffer* **TableType**

Public Functions

inline **Trigger**()

inline **Trigger**(int64_t _timestamp, *TriggerType* _type)

Public Members

int64_t **timestamp**

TriggerType **type**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

struct **TriggerBuilder**

Public Functions

inline void **add_timestamp**(int64_t timestamp)

inline void **add_type**(*TriggerType* type)

inline explicit **TriggerBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

TriggerBuilder &**operator=**(const *TriggerBuilder*&)

inline *flatbuffers*::Offset<*TriggerFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **TriggerFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *Trigger* **NativeTableType**

Public Functions

inline int64_t **timestamp**() const
Timestamp (µs).

inline *TriggerType* **type**() const
Type of trigger that occurred.

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *Trigger* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*Trigger* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*Trigger* *_o, const *TriggerFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*TriggerFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Trigger* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

struct **TriggerPacket** : public *flatbuffers*::NativeTable

Public Types

typedef *TriggerPacketFlatbuffer* **TableType**

Public Functions

inline **TriggerPacket**()

inline **TriggerPacket**(const *dv*::*cvector*<*Trigger*> &_elements)

Public Members

dv::*cvector*<*Trigger*> **elements**

Public Static Functions

static inline constexpr const char ***GetFullyQualifiedName**()

Friends

inline friend *std*::ostream &**operator<<**(*std*::ostream &os, const *TriggerPacket* &packet)

struct **TriggerPacketBuilder**

Public Functions

inline void **add_elements**(*flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*TriggerFlatbuffer*>>> elements)

inline explicit **TriggerPacketBuilder**(*flatbuffers*::FlatBufferBuilder &_fbb)

TriggerPacketBuilder &**operator=**(const *TriggerPacketBuilder*&)

inline *flatbuffers*::Offset<*TriggerPacketFlatbuffer*> **Finish**()

Public Members

flatbuffers::FlatBufferBuilder &**fbb_**

flatbuffers::uoffset_t **start_**

struct **TriggerPacketFlatbuffer** : private *flatbuffers*::Table

Public Types

typedef *TriggerPacket* **NativeTableType**

Public Functions

inline const *flatbuffers*::Vector<*flatbuffers*::Offset<*TriggerFlatbuffer*>> ***elements**() const

inline bool **Verify**(*flatbuffers*::Verifier &verifier) const

inline *TriggerPacket* ***UnPack**(const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

inline void **UnPackTo**(*TriggerPacket* *_o, const *flatbuffers*::resolver_function_t *_resolver = nullptr) const

Public Static Functions

static inline const *flatbuffers*::TypeTable ***MiniReflectTypeTable**()

static inline constexpr const char ***GetFullyQualifiedName**()

static inline void **UnPackToFrom**(*TriggerPacket* *_o, const *TriggerPacketFlatbuffer* *_fb, const *flatbuffers*::resolver_function_t *_resolver = nullptr)

static inline *flatbuffers*::Offset<*TriggerPacketFlatbuffer*> **Pack**(*flatbuffers*::FlatBufferBuilder &_fbb, const *TriggerPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

Public Static Attributes

static constexpr const char ***identifier** = "TRIG"

struct **Type**

Public Functions

inline constexpr **Type**() noexcept

inline constexpr **Type**(const *std*::string_view identifier_, const size_t sizeOfType_, *PackFuncPtr* pack_, *UnpackFuncPtr* unpack_, *ConstructPtr* construct_, *DestructPtr* destruct_, *TimeElementExtractorPtr* timeElementExtractor_, *TimeRangeExtractorPtr* timeRangeExtractor_)

~Type() = default

Type(const *Type* &t) = default

Type &**operator=**(const *Type* &rhs) = default

Type(*Type* &&t) = default

Type &**operator=**(*Type* &&rhs) = default

inline constexpr bool **operator==**(const *Type* &rhs) const noexcept

inline constexpr bool **operator!=**(const *Type* &rhs) const noexcept

Public Members

uint32_t **id**

size_t **sizeOfType**

PackFuncPtr **pack**

UnpackFuncPtr **unpack**

ConstructPtr **construct**

DestructPtr **destruct**

TimeElementExtractorPtr **timeElementExtractor**

TimeRangeExtractorPtr **timeRangeExtractor**

struct **TypedObject**

Public Functions

inline constexpr **TypedObject**(const *Type* &type_)

inline **~TypedObject**() noexcept

TypedObject(const *TypedObject* &t) = delete

TypedObject &**operator=**(const *TypedObject* &rhs) = delete

inline **TypedObject**(*TypedObject* &&t)

inline *TypedObject* &**operator=**(*TypedObject* &&rhs)

inline constexpr bool **operator==**(const *TypedObject* &rhs) const noexcept

inline constexpr bool **operator!=**(const *TypedObject* &rhs) const noexcept

template<class **TargetType**>

inline *std*::shared_ptr<*TargetType*> **moveToSharedPtr**()

Cast and move the pointer to the data into a shared pointer. The underlying data is not affected, but it invalidates this instance and passes the ownership of the data to the shared pointer - it will take care of memory management from the point of this method call.

Template Parameters
TargetType – Target type to cast the typed obect into

Returns

Public Members

void ***obj**

Type **type**

struct **TypeError**

Public Types

using **Info** = *dv*::*cstring*

Public Static Functions

static inline *std*::string **format**(const *Info* &info)

class **UNIXSocket** : public *dv*::*io*::*network*::*SocketBase*

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/network/unix_socket.hpp> Minimal wrapper of UNIX socket. It follows RAII principle, the socket will closed and released when this object is released.

Public Functions

inline explicit **UNIXSocket**(*asioUNIX*::socket &&s)

Initial a socket wrapper by taking ownership of a connected socket.

Parameters
s –

inline **~UNIXSocket**() override

inline virtual bool **isOpen**() const override

Check whether socket is open and active.

Returns
True if socket is open, false otherwise.

inline virtual void **close**() override

Close underlying UNIX socket cleanly.

inline virtual void **write**(const asio::const_buffer &buf, CompletionHandler &&wrHandler) override

Write handler needs following signature: void (const boost::system::error_code &, size_t)

inline virtual void **read**(const asio::mutable_buffer &buf, CompletionHandler &&rdHandler) override

Read handler needs following signature: void (const boost::system::error_code &, size_t)

inline virtual void **syncWrite**(const asio::const_buffer &buf) override

Blocking write data to the socket.

Parameters
buf – Data to write.

inline virtual void **syncRead**(const asio::mutable_buffer &buf) override

Blocking read from socket.

Parameters
buf – Buffer for data to be read into.

Private Members

asioUNIX::socket **socket**

bool **socketClosed** = false

class **UpdateIntervalOrFeatureCountRedetection** : public *dv*::*features*::*RedetectionStrategy*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/redetection_strategy.hpp> Redetection strategy based on interval from last detection or minimum number of tracks. This class combines redetection logic from *UpdateIntervalRedetection* and *FeatureCountRedetection*.

Public Functions

inline explicit **UpdateIntervalOrFeatureCountRedetection**(const *dv*::*Duration* updateInterval, const float minimumProportionOfTracks)

Redetection strategy based on updating if specific amount of time from last detection has passed or minimum number of tracks to follow.

inline virtual bool **decideRedetection**(const *TrackerBase* &tracker) override

Check whether to perform redetection.

Private Members

UpdateIntervalRedetection **updateIntervalRedetection**

FeatureCountRedetection **featureCountRedetection**

class **UpdateIntervalRedetection** : public *dv*::*features*::*RedetectionStrategy*

#include </builds/inivation/dv/dv-processing/include/dv-processing/features/redetection_strategy.hpp> Redetection strategy based on interval from last detection.

Public Functions

inline explicit **UpdateIntervalRedetection**(const *dv*::*Duration* updateInterval)

Redetection strategy based on updating if specific amount of time from last detection has passed.

inline virtual bool **decideRedetection**(const *TrackerBase* &tracker) override

Check whether to perform redetection.

Protected Attributes

const int64_t **mUpdateIntervalTime**

int64_t **mLastDetectionTime** = -*std*::numeric_limits<int64_t>::infinity()

struct **WriteJob**

Public Functions

inline **WriteJob**(const asio::const_buffer &buffer, *SocketBase*::*CompletionHandler* handler)

Public Members

asio::const_buffer **mBuffer**

SocketBase::*CompletionHandler* **mHandler**

class **WriteOnlyFile** : private *dv*::*io*::*SimpleWriteOnlyFile*

Public Functions

WriteOnlyFile() = delete

inline **WriteOnlyFile**(const *std*::filesystem::path &filePath, const *std*::string_view outputInfo,
std::unique_ptr<*dv*::*io*::*compression*::*CompressionSupport*> compression,
std::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> stats = nullptr)

inline **WriteOnlyFile**(const *std*::filesystem::path &filePath, const *std*::string_view outputInfo, const
CompressionType compression = *CompressionType*::*NONE*,
std::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> stats = nullptr)

inline **~WriteOnlyFile**()

inline void **write**(const *dv*::*types*::*TypedObject* *const packet, const int32_t streamId)

inline void **write**(const void *ptr, const *dv*::*types*::*Type* &type, const int32_t streamId)

Private Functions

inline void **pushVersion**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> version)

inline void **pushHeader**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> header)

inline void **pushPacket**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> packet)

inline void **pushFileDataTable**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> fileDataTable)

inline void **writeThread**()

inline void **stop**()

inline void **emptyWriteBuffer**()

inline void **writeVersion**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> packet)

inline void **writeHeader**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> packet)

inline void **writePacket**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> packet)

inline void **writeFileDataTable**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> packet)

Private Members

std::string **mOutputInfo**

dv::*io*::*Writer* **mWriter**

std::mutex **mMutex**

std::queue<*std*::function<void(void)>> **mWriteBuffer**

std::atomic<bool> **mStopRequested** = {false}

std::thread **mWriteThread**

class **WriteOrderedSocket**

#include </builds/inivation/dv/dv-processing/include/dv-processing/io/network/write_ordered_socket.hpp> Write ordered socket. Implemented because in asio simultaneous async_writes are not allowed.

See also:

https://stackoverflow.com/questions/45813835/boostasio-ordering-of-data-sent-to-a-socket

Public Functions

inline explicit **WriteOrderedSocket**(*std*::unique_ptr<*SocketBase*> &&socket)

inline void **write**(const asio::const_buffer &buf, *SocketBase*::*CompletionHandler* &&wrHandler)

Add a buffer to be written out to the socket. This call adds the buffer to a ordered queue that guarantees that will chain multiple write_async calls to the socket so no simultaneous calls would happen.

Parameters

- **buf** – Buffers to be written into the socket.
- **wrHandler** – Write handler that is called when buffer write is completed.

inline void **close**()

Close the underlying socket.

inline bool **isOpen**() const

Check whether underlying socket is open

Returns

inline void **read**(const asio::mutable_buffer &buf, *SocketBase*::*CompletionHandler* &&rdHandler)

Read data from the socket. This only wraps the read call of the underlying socket.

Parameters

- **buf** –
- **rdHandler** –

Private Members

std::deque<*WriteJob*> **mWriteQueue**

No locking for writeQueue because all changes are posted to io_service thread.

std::unique_ptr<*dv*::*io*::*network*::*SocketBase*> **mSocket**

Underlying socket.

class **Writer**

Public Types

using **WriteHandler** = *dv*::*std_function_exact*<void(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*>)>

Public Functions

Writer() = delete

inline explicit **Writer**(*std*::unique_ptr<*dv*::*io*::*compression*::*CompressionSupport*> compression, *std*::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> stats = nullptr, *std*::unique_ptr<*dv*::*FileDataTable*> dataTable = nullptr)

inline explicit **Writer**(const *dv*::*CompressionType* compression, *std*::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> stats = nullptr, *std*::unique_ptr<*dv*::*FileDataTable*> dataTable = nullptr)

~Writer() = default

Writer(const *Writer* &other) = delete

Writer &**operator=**(const *Writer* &other) = delete

Writer(*Writer* &&other) noexcept = default

Writer &**operator=**(*Writer* &&other) noexcept = default

inline auto **getCompressionType**()

inline size_t **writeAedatVersion**(const *WriteHandler* &writeHandler)

inline size_t **writeHeader**(const int64_t dataTablePosition, const *std*::string_view infoNode, const *WriteHandler* &writeHandler)

inline size_t **writePacket**(const *dv*::*types*::*TypedObject* *const packet, const int32_t streamId, const *WriteHandler* &writeHandler)

inline size_t **writePacket**(const void *ptr, const *dv*::*types*::*Type* &type, const int32_t streamId, const *WriteHandler* &writeHandler)

inline int64_t **writeFileDataTable**(const *WriteHandler* &writeHandler)

Public Static Functions

static inline *std*::shared_ptr<*dv*::*io*::*support*::*IODataBuffer*> **encodeAedat4Version**()

static inline *std*::shared_ptr<*dv*::*io*::*support*::*IODataBuffer*> **encodeFileHeader**(const int64_t dataTablePosition, const *std*::string_view infoNode, const *dv*::*CompressionType* compressionType)

static inline void **encodePacketHeader**(const *std*::shared_ptr<*dv*::*io*::*support*::*IODataBuffer*> packet, const int32_t streamId)

static inline *std*::shared_ptr<*dv*::*io*::*support*::*IODataBuffer*> **encodePacketBody**(const void *ptr, const *dv*::*types*::*Type* &type)

static inline *std*::shared_ptr<*dv*::*io*::*support*::*IODataBuffer*> **encodeFileDataTable**(const *dv*::*FileDataTable* &table)

Private Functions

inline void **writeToDestination**(const *std*::shared_ptr<const *dv*::*io*::*support*::*IODataBuffer*> data, const *WriteHandler* &writeHandler)

inline void **compressData**(*dv*::*io*::*support*::*IODataBuffer* &packet)

inline void **updateFileDataTable**(const uint64_t byteOffset, const uint64_t numElements, const int64_t timestampStart, const int64_t timestampEnd, const *dv*::PacketHeader &header)

Private Members

std::unique_ptr<*dv*::*io*::*support*::*IOStatistics*> **mStats**

std::unique_ptr<*dv*::*io*::*compression*::*CompressionSupport*> **mCompressionSupport**

std::unique_ptr<*dv*::*FileDataTable*> **mFileDataTable**

uint64_t **mByteOffset** = {0}

class **XMLConfigReader**

Public Functions

XMLConfigReader() = delete

inline **XMLConfigReader**(const *std*::string_view xmlContent)

inline **XMLConfigReader**(const *std*::string_view xmlContent, const *std*::string_view expectedRootName)

inline const *XMLTreeNode* &**getRoot**() const

Private Functions

inline void **parseXML**(const *std*::string_view xmlContent, const *std*::string_view expectedRootName)

Private Members

XMLTreeNode **mRoot**

Private Static Functions

static inline *std*::vector<*std*::reference_wrapper<const boost::property_tree::ptree>> **xmlFilterChildNodes**(const boost::property_tree::ptree &content, const *std*::string &name)

static inline void **consumeXML**(const boost::property_tree::ptree &content, *XMLTreeNode* &node)

static inline *dv*::*io*::*support*::*VariantValueOwning* **stringToValueConverter**(const *std*::string &typeStr, const *std*::string &valueStr)

class **XMLConfigWriter**

Public Functions

XMLConfigWriter() = delete

inline **XMLConfigWriter**(const *XMLTreeNode* &root)

inline const *std*::string &**getXMLContent**() const

Private Functions

inline void **writeXML**(const *XMLTreeNode* &root)

Private Members

std::string **mXMLOutputContent**

Private Static Functions

static inline boost::property_tree::ptree **generateXML**(const *XMLTreeNode* &node, const *std*::string &prevPath)

static inline *std*::pair<*std*::string, *std*::string> **valueToStringConverter**(const *dv*::*io*::*support*::*VariantValueOwning* &value)

struct **XMLTreeAttribute** : public *dv*::*io*::*support*::*XMLTreeCommon*

Public Functions

XMLTreeAttribute() = delete

inline explicit **XMLTreeAttribute**(const *std*::string_view name)

Public Members

dv::*io*::*support*::*VariantValueOwning* **mValue**

struct **XMLTreeCommon**

Subclassed by *dv::io::support::XMLTreeAttribute*, *dv::io::support::XMLTreeNode*

Public Functions

XMLTreeCommon() = delete

inline explicit **XMLTreeCommon**(const *std*::string_view name)

inline bool **operator==**(const *XMLTreeCommon* &rhs) const noexcept

inline auto **operator<=>**(const *XMLTreeCommon* &rhs) const noexcept

inline bool **operator==**(const *std*::string_view &rhs) const noexcept

inline auto **operator<=>**(const *std*::string_view &rhs) const noexcept

Public Members

std::string **mName**

struct **XMLTreeNode** : public *dv*::*io*::*support*::*XMLTreeCommon*

Public Functions

inline explicit **XMLTreeNode**()

inline explicit **XMLTreeNode**(const *std*::string_view name)

Public Members

std::vector<*XMLTreeNode*> **mChildren**

std::vector<*XMLTreeAttribute*> **mAttributes**

class **ZstdCompressionSupport** : public *dv*::*io*::*compression*::*CompressionSupport*

Public Functions

inline explicit **ZstdCompressionSupport**(const *CompressionType* type)

inline explicit **ZstdCompressionSupport**(const int compressionLevel)

Create a Zstd compression support class with custom compression. Internally sets compression type to `CompressionType::ZSTD`.

See also:

For more info on compression level values see here: https://facebook.github.io/zstd/zstd_manual.html

Parameters
compressionLevel – Compression level, recommended range is [1, 22].

inline virtual void **compress**(*dv*::*io*::*support*::*IODataBuffer* &packet) override

Private Members

std::shared_ptr<ZSTD_CCtx_s> **mContext**

int **mLevel** = {3}

class **ZstdDecompressionSupport** : public *dv*::*io*::*compression*::*DecompressionSupport*

Public Functions

inline explicit **ZstdDecompressionSupport**(const *CompressionType* type)

inline virtual void **decompress**(*std*::vector<*std*::byte> &src, *std*::vector<*std*::byte> &target) override

Private Functions

inline void **initDecompressionContext**()

Private Members

std::shared_ptr<ZSTD_DCtx_s> **mContext**

template<class **T**>
concept **MeanShiftKernel**

template<class **T1**, class **T2**>
concept **Accepts**

template<class **T**>
concept **AddressableEvent**

template<class **T**>

concept **BlockAccessible**

template<class **Type**>
concept **CompatibleWithSlicer**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/stream_slicer.hpp> Concept that verifies that given type is compatible for use with stream slicer.

tparam Type
Type to verify

template<class **T**>
concept **Coordinate2D**

template<class **T**>
concept **Coordinate2DAccessors**

template<class **T**>
concept **Coordinate2DCostructible**

template<class **T**>
concept **Coordinate2DIterable**

template<class **T**>
concept **Coordinate2DMembers**

template<class **T**>
concept **Coordinate2DMutableIterable**

template<class **T**>
concept **Coordinate3D**

template<class **T**>
concept **Coordinate3DAccessors**

template<class **T**>
concept **Coordinate3DCostructible**

template<class **T**>
concept **Coordinate3DIterable**

template<class **T**>
concept **Coordinate3DMembers**

template<class **T**>
concept **Coordinate3DMutableIterable**

template<class **Packet**>
concept **DataPacket**

template<class **T**, class **Input**>
concept **DVFeatureDetectorAlgorithm**

template<class **T**>
concept **EigenType**

template<class **T**>
concept **Enum**

template<class **T**, class **EventStoreType**>

concept **EventFilter**

template<class **T**, class **EventStoreType**>
concept **EventOutputGenerator**

template<class **T**>
concept **EventStorage**

template<class **T**, class **EventStoreType**>
concept **EventToEventConverter**

template<class **T**, class **EventStoreType**>
concept **EventToFrameConverter**

template<class **T**, class **Input**>
concept **FeatureDetectorAlgorithm**

template<class **T**>
concept **FlatbufferPacket**

template<class **T**>
concept **FrameOutputGenerator**

template<class **T**, class **EventStoreType**>
concept **FrameToEventConverter**

template<class **T**>
concept **FrameToFrameConverter**

template<class **T**>
concept **HasElementsVector**

template<class **T**>
concept **HasTimestampedElementsVector**

template<class **T**>
concept **HasTimestampedElementsVectorByAccessor**

template<class **T**>
concept **HasTimestampedElementsVectorByMember**

template<class **T1**, class **T2**>
concept **InputStreamableFrom**

template<class **T1**, class **T2**>
concept **InputStreamableTo**

template<class **T**>
concept **OutputStreamable**

template<typename **FUNC**, typename **RETURN_TYPE**, typename ...**ARGUMENTS_TYPES**>
concept **InvocableReturnArgumentsStrong**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/concepts.hpp> Checks if function is invocable with the given argument types exactly and its return value is the same as the given return type.

tparam FUNC
function-like object to check.

tparam RETURN_TYPE
required return type.

tparam ARGUMENTS_TYPES
required argument types.

template<typename **FUNC**, typename **RETURN_TYPE**, typename ...**ARGUMENTS_TYPES**>
concept **InvocableReturnArgumentsWeak**

#include </builds/inivation/dv/dv-processing/include/dv-processing/core/concepts.hpp> Checks if function is invocable with the given argument types and its return value is convertible to the given return type.

tparam FUNC
function-like object to check.

tparam RETURN_TYPE
required return type.

tparam ARGUMENTS_TYPES
required argument types.

template<class **T1**, class **T2**>
concept **IOStreamableFrom**

template<class **T1**, class **T2**>
concept **IOStreamableTo**

template<typename **T**>
concept **Iterable**

template<class **T**>
concept **KeyPointVector**

template<typename **T**>
concept **MutableIterable**

template<typename **T**>
concept **number**

template<class **T**>
concept **OpenCVFeatureDetectorAlgorithm**

template<class **T1**, class **T2**>
concept **OutputStreamableFrom**

template<class **T1**, class **T2**>
concept **OutputStreamableTo**

template<class **T**>
concept **SupportsConstantDepth**

template<class **T**>
concept **TimedImageContainer**

template<class **T**>
concept **Timestamped**

template<class **T**>
concept **TimestampedByAccessor**

template<class **T**>
concept **TimestampedByMember**

template<class **T**>

concept **TimestampedIterable**

template<class **T**>
concept **TimestampMatrixContainer**

template<class **T**, class **EventStoreType**>
concept **TimeSurface**

template<typename **T**>
concept **HasCustomExceptionFormatter**

template<typename **T**>
concept **HasExtraExceptionInfo**

template<typename **T**>
concept **NoCustomExceptionFormatter**

namespace **dv**

Typedefs

using **EventStoreIterator** = *AddressableEventStorageIterator*<*dv*::Event, *dv*::*EventPacket*>

using **EventStore** = *AddressableEventStorage*<*dv*::Event, *dv*::*EventPacket*>

using **DepthEventStore** = *dv*::*AddressableEventStorage*<*dv*::DepthEvent, *dv*::*DepthEventPacket*>

using **EventStreamSlicer** = StreamSlicer<*EventStore*>

using **FrameStreamSlicer** = StreamSlicer<*dv*::*cvector*<*dv*::*Frame*>>

using **IMUStreamSlicer** = StreamSlicer<*dv*::*cvector*<*dv*::*IMU*>>

using **TriggerStreamSlicer** = StreamSlicer<*dv*::*cvector*<*dv*::*Trigger*>>

using **TimeSurface** = *TimeSurfaceBase*<*EventStore*>

using **SpeedInvariantTimeSurface** = *SpeedInvariantTimeSurfaceBase*<*EventStore*>

using **PixelAccumulator** = *EdgeMapAccumulator*

using **StereoEventStreamSlicer** = *AddressableStereoEventStreamSlicer*<*dv*::*EventStore*>

using **TimestampClock** = *std*::chrono::system_clock

using **TimestampResolution** = *std*::chrono::microseconds

using **Duration** = *TimestampResolution*

Duration type that stores microsecond time period.

using **TimePoint** = *std*::chrono::time_point<*TimestampClock*, *TimestampResolution*>

Timepoint type that stores microsecond time point related to system clock.

using **cstring** = *basic_cstring*<char>

using **cwstring** = *basic_cstring*<wchar_t>

using **cu8string** = *basic_cstring*<char8_t>

using **cu16string** = *basic_cstring*<char16_t>

using **cu32string** = *basic_cstring*<char32_t>

Enums

enum **EventColor**

The EventColor enum contains the color of the Bayer color filter for a specific event address. WHITE means White/No Filter. Please take into account that there are usually twice as many green pixels as there are red or blue ones.

Values:

enumerator **WHITE**

enumerator **RED**

enumerator **GREEN**

enumerator **BLUE**

enum **PixelArrangement**

Color pixel block arrangement on the sensor. The sensor usually contain one red, one blue, and two green pixels. They can be arranged in different order, so exact color extraction, the pixel arrangement needs to be known.

Values:

enumerator **RGBG**

enumerator **GRGB**

enumerator **GBGR**

enumerator **BGRG**

enum class **TimeSlicingApproach**

Time handling approaches for number based slicing.

Values:

enumerator **BACKWARD**

Assign gap elements between previous numeric slice and current one.

enumerator **FORWARD**

Assign gap elements between current numeric slice and next one.

enum class **FrameFormat** : int8_t

Format values are compatible with OpenCV. Pixel layout follows OpenCV standard.

Values:

enumerator **GRAY**

enumerator **OPENCV_8U_C1**

enumerator **OPENCV_8S_C1**

enumerator **OPENCV_16U_C1**

enumerator **OPENCV_16S_C1**

enumerator **OPENCV_32S_C1**

enumerator **OPENCV_32F_C1**

enumerator **OPENCV_64F_C1**

enumerator **OPENCV_16F_C1**

enumerator **OPENCV_8U_C2**

enumerator **OPENCV_8S_C2**

enumerator **OPENCV_16U_C2**

enumerator **OPENCV_16S_C2**

enumerator **OPENCV_32S_C2**

enumerator **OPENCV_32F_C2**

enumerator **OPENCV_64F_C2**

enumerator **OPENCV_16F_C2**

enumerator **BGR**

enumerator **OPENCV_8U_C3**

enumerator **OPENCV_8S_C3**

enumerator **OPENCV_16U_C3**

enumerator **OPENCV_16S_C3**

enumerator **OPENCV_32S_C3**

enumerator **OPENCV_32F_C3**

enumerator **OPENCV_64F_C3**

enumerator **OPENCV_16F_C3**

enumerator **BGRA**

enumerator **OPENCV_8U_C4**

enumerator **OPENCV_8S_C4**

enumerator **OPENCV_16U_C4**

enumerator **OPENCV_16S_C4**

enumerator **OPENCV_32S_C4**

enumerator **OPENCV_32F_C4**

enumerator **OPENCV_64F_C4**

enumerator **OPENCV_16F_C4**

enumerator **MIN**

enumerator **MAX**

enum class **FrameSource** : int8_t

Image data source.

Values:

enumerator **UNDEFINED**

Undefined source, this value indicates that source field shouldn't be considered at all.

enumerator **SENSOR**

enumerator **ACCUMULATION**

enumerator **MOTION_COMPENSATION**

enumerator **SYNTHETIC**

enumerator **RECONSTRUCTION**

enumerator **VISUALIZATION**

enumerator **OTHER**

enumerator **MIN**

enumerator **MAX**

enum class **TriggerType** : int8_t

Values:

enumerator **TIMESTAMP_RESET**

A timestamp reset occurred.

enumerator **EXTERNAL_SIGNAL_RISING_EDGE**

enumerator **EXTERNAL_SIGNAL_FALLING_EDGE**

enumerator **EXTERNAL_SIGNAL_PULSE**

enumerator **EXTERNAL_GENERATOR_RISING_EDGE**

enumerator **EXTERNAL_GENERATOR_FALLING_EDGE**

enumerator **APS_FRAME_START**

enumerator **APS_FRAME_END**

enumerator **APS_EXPOSURE_START**

enumerator **APS_EXPOSURE_END**

enumerator **MIN**

enumerator **MAX**

enum class **Constants** : int32_t

Values:

enumerator **AEDAT_VERSION_LENGTH**

enumerator **MIN**

enumerator **MAX**

enum class **CompressionType** : int32_t

Values:

enumerator **NONE**

enumerator **LZ4**

enumerator **LZ4_HIGH**

enumerator **ZSTD**

enumerator **ZSTD_HIGH**

enumerator **MIN**

enumerator **MAX**

Functions

inline void **runtime_assert**(const bool expression, const *std*::string_view message, const *std*::source_location &location = *std*::source_location::current())

inline uint32_t **coordinateHash**(const int16_t x, const int16_t y)

Function that creates perfect hash for 2d coordinates.

Parameters

- **x** – x coordinate
- **y** – y coordinate

Returns

a 64 bit hash that uniquely identifies the coordinates

template<class **EventStoreType**>
inline void **roiFilter**(const *EventStoreType* &in, *EventStoreType* &out, const cv::Rect &roi)

Extracts only the events that are within the defined region of interest. This function copies the events from the in EventStore into the given out EventStore, if they intersect with the given region of interest rectangle.

Parameters

- **in** – The EventStore to operate on. Won't be modified.
- **out** – The EventStore to put the ROI events into. Will get modified.
- **roi** – The rectangle with the region of interest.

template<class **EventStoreType**>
inline void **polarityFilter**(const *EventStoreType* &in, *EventStoreType* &out, bool polarity)

Filters events by polarity. Only events that exhibit the same polarity as given in polarity are kept.

Parameters

- **in** – Incoming EventStore to operate on. Won't get modified.
- **out** – The outgoing EventStore to store the kept events on
- **polarity** – The polarity of the events that should be kept

template<class **EventStoreType**>
inline void **maskFilter**(const *EventStoreType* &in, *EventStoreType* &out, const cv::Mat &mask)

Filter event with a coordinate mask. Discards any events that happen on coordinates where mask has a zero value and retains all events with coordinates where mask has a non-zero value.

Template Parameters

EventStoreType – Class for the event store container.

Parameters

- **in** – Incoming EventStore to operate on. Won't get modified.
- **out** – The outgoing EventStore to store the kept events on
- **mask** – The mask to be applied (requires CV_8UC1 type).

template<class **EventStoreType**>
inline void **scale**(const *EventStoreType* &in, *EventStoreType* &out, double xDivision, double yDivision)

Projects the event coordinates onto a smaller range. The x- and y-coordinates the divided by xFactor and yFactor respectively and floored to the next integer. This forms the new coordinates of the event. Due to the nature of this, it can happen that multiple events end up happening simultaneously at the same location. This

is still a valid event stream, as time keeps monotonically increasing, but is something that is unlikely to be generated by an event camera.

Parameters

- **in** – The EventStore to operate on. Won't get modified
- **out** – The outgoing EventStore to store the projected events on
- **xDivision** – Division factor for the x-coordinate for the events
- **yDivision** – Division factor for the y-coordinate of the events

template<class **EventStoreType**>
inline cv::Rect **boundingRect**(const *EventStoreType* &packet)

Computes and returns a rectangle with dimensions such that all the events in the given `EventStore` fall into the bounding box.

Parameters
packet – The EventStore to work on

Returns
The smallest possible rectangle that contains all the events in packet.

inline *EventColor* **colorForEvent**(const Event &evt, const *PixelArrangement* arrangement = *PixelArrangement*::*RGBG*)

Determine the color of the Bayer color filter for a specific event, based on its address. Please take into account that there are usually twice as many green pixels as there are red or blue ones.

Parameters

- **evt** – event to determine filter color for.
- **pixelArrangement** – color pixel arrangement for a sensor.

Returns
filter color.

inline *TimePoint* **toTimePoint**(const int64_t timestamp)

Convert a 64-bit integer microsecond timestamp into a chrono time-point.

Parameters
timestamp – 64-bit integer microsecond timestamp

Returns
Chrono time point (microseconds, system clock).

inline int64_t **fromTimePoint**(const *TimePoint* timepoint)

Convert a chrono time-point into a 64-bit integer microsecond timestamp.

Parameters
timestamp – Chrono time point (microseconds, system clock).

Returns
64-bit integer microsecond timestamp

inline int64_t **now**()

Returns
Current system clock timestamp in microseconds as 64-bit integer.

template<*dv*::*concepts*::*Enum* **Enumeration**>

constexpr *std*::underlying_type_t<*Enumeration*> **EnumAsInteger**(const *Enumeration* value) noexcept

Functions to help handle enumerations and their values.

template<*dv*::*concepts*::*Enum* **Enumeration**, *std*::integral **T**>
constexpr *Enumeration* **IntegerAsEnum**(const *T* value) noexcept

template<typename **T**, typename **U**>
inline bool **vectorContains**(const *std*::vector<*T*> &vec, const *U* &item)

Functions to help deal with common vector operations: bool vectorContains(vec, item) bool vectorContainsIf(vec, predicate) bool vectorRemove(vec, item) bool vectorRemoveIf(vec, predicate) void vectorSortUnique(vec) void vectorSortUnique(vec, comparator)

template<typename **T**, typename **Pred**>
inline bool **vectorContainsIf**(const *std*::vector<*T*> &vec, *Pred* predicate)

template<typename **T**, typename **U**>
inline size_t **vectorRemove**(*std*::vector<*T*> &vec, const *U* &item)

template<typename **T**, typename **Pred**>
inline size_t **vectorRemoveIf**(*std*::vector<*T*> &vec, *Pred* predicate)

template<typename **T**>
inline void **vectorSortUnique**(*std*::vector<*T*> &vec)

template<typename **T**, typename **Compare**>
inline void **vectorSortUnique**(*std*::vector<*T*> &vec, *Compare* comp)

inline *std*::filesystem::path **pathResolveNonExisting**(const *std*::filesystem::path &path)

Path cleanup functions for existing paths (canonical) and possibly non-existing ones (absolute).

inline *std*::filesystem::path **pathResolveExisting**(const *std*::filesystem::path &path)

template<typename **ObjectT**, typename ...**Args**>
inline void ***mallocConstructorSize**(const size_t sizeOfObject, *Args*&&... args)

template<typename **ObjectT**, typename ...**Args**>
inline void ***mallocConstructor**(*Args*&&... args)

template<typename **ObjectT**>
inline void **mallocDestructor**(void *object) noexcept

inline *std*::string **errnoToString**(int errorNumber)

template<*concepts*::*Coordinate2D* **Input**>
inline bool **isWithinDimensions**(const *Input* &point, const cv::Size &resolution)

Check whether given point is non-negative and within dimensions of given resolution. The following check is performed: X ∈ [0; (width - 1)] and Y ∈ [0; (height - 1)]. Function will check floating point coordinate fractional part overflow, it will return false in case even fractional part is beyond the valid range.

Parameters

- **point** – Coordinates to check.
- **resolution** – Pixel space resolution.

Returns

True if coordinates are within valid range, false otherwise.

inline bool **operator==**(const *BoundingBox* &lhs, const *BoundingBox* &rhs)

inline bool **operator==**(const *BoundingBoxPacket* &lhs, const *BoundingBoxPacket* &rhs)

inline const *flatbuffers*::TypeTable ***BoundingBoxTypeTable**()

inline const *flatbuffers*::TypeTable ***BoundingBoxPacketTypeTable**()

inline *flatbuffers*::Offset<*BoundingBoxFlatbuffer*> **CreateBoundingBox**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, float topLeftX = 0.0f, float topLeftY = 0.0f, float bottomRightX = 0.0f, float bottomRightY = 0.0f, float confidence = 0.0f, *flatbuffers*::Offset<*flatbuffers*::String> label = 0)

inline *flatbuffers*::Offset<*BoundingBoxFlatbuffer*> **CreateBoundingBoxDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, float topLeftX = 0.0f, float topLeftY = 0.0f, float bottomRightX = 0.0f, float bottomRightY = 0.0f, float confidence = 0.0f, const char *label = nullptr)

inline *flatbuffers*::Offset<*BoundingBoxFlatbuffer*> **CreateBoundingBox**(*flatbuffers*::FlatBufferBuilder &_fbb, const *BoundingBox* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*BoundingBoxPacketFlatbuffer*> **CreateBoundingBoxPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<*flat*
elements = 0)

inline *flatbuffers*::Offset<*BoundingBoxPacketFlatbuffer*> **CreateBoundingBoxPacketDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<*flatbuffers*::Offset *elements = nullptr)

inline *flatbuffers*::Offset<*BoundingBoxPacketFlatbuffer*> **CreateBoundingBoxPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, const *BoundingBoxPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*BoundingBoxPacketFlatbuffer* ***GetBoundingBoxPacket*** (const void *buf)

inline const *dv*::*BoundingBoxPacketFlatbuffer* ***GetSizePrefixedBoundingBoxPacket*** (const void *buf)

inline const char ***BoundingBoxPacketIdentifier*** ()

inline bool ***BoundingBoxPacketBufferHasIdentifier*** (const void *buf)

inline bool ***VerifyBoundingBoxPacketBuffer*** (*flatbuffers*::Verifier &verifier)

inline bool ***VerifySizePrefixedBoundingBoxPacketBuffer*** (*flatbuffers*::Verifier &verifier)

inline void ***FinishBoundingBoxPacketBuffer*** (*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*BoundingBoxPacketFlatbuffer*> root)

inline void ***FinishSizePrefixedBoundingBoxPacketBuffer*** (*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*BoundingBoxPacketFlatbuffer*> root)

inline *std*::unique_ptr<*BoundingBoxPacket*> ***UnPackBoundingBoxPacket*** (const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool ***operator==*** (const DepthEvent &lhs, const DepthEvent &rhs)

inline bool ***operator==*** (const *DepthEventPacket* &lhs, const *DepthEventPacket* &rhs)

inline const *flatbuffers*::TypeTable ***DepthEventTypeTable*** ()

inline const *flatbuffers*::TypeTable ***DepthEventPacketTypeTable*** ()

FLATBUFFERS_MANUALLY_ALIGNED_STRUCT (8) DepthEvent final

FLATBUFFERS_STRUCT_END (DepthEvent, 16)

inline *flatbuffers*::Offset<*DepthEventPacketFlatbuffer*> ***CreateDepthEventPacket*** (*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<const DepthEvent*>> elements = 0)

inline *flatbuffers*::Offset<*DepthEventPacketFlatbuffer*> ***CreateDepthEventPacketDirect*** (*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<DepthEvent> *elements = nullptr)

inline *flatbuffers*::Offset<*DepthEventPacketFlatbuffer*> **CreateDepthEventPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, const *DepthEventPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*DepthEventPacketFlatbuffer* *__**GetDepthEventPacket**__(const void *buf)

inline const *dv*::*DepthEventPacketFlatbuffer* *__**GetSizePrefixedDepthEventPacket**__(const void *buf)

inline const char *__**DepthEventPacketIdentifier**__()

inline bool __**DepthEventPacketBufferHasIdentifier**__(const void *buf)

inline bool __**VerifyDepthEventPacketBuffer**__(*flatbuffers*::Verifier &verifier)

inline bool __**VerifySizePrefixedDepthEventPacketBuffer**__(*flatbuffers*::Verifier &verifier)

inline void __**FinishDepthEventPacketBuffer**__(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*DepthEventPacketFlatbuffer*> root)

inline void __**FinishSizePrefixedDepthEventPacketBuffer**__(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*DepthEventPacketFlatbuffer*> root)

inline *std*::unique_ptr<*DepthEventPacket*> **UnPackDepthEventPacket**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool __**operator==**__(const *DepthFrame* &lhs, const *DepthFrame* &rhs)

inline const *flatbuffers*::TypeTable *__**DepthFrameTypeTable**__()

inline *flatbuffers*::Offset<*DepthFrameFlatbuffer*> **CreateDepthFrame**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, int16_t sizeX = 0, int16_t sizeY = 0, uint16_t minDepth = 0, uint16_t maxDepth = 65535, uint16_t step = 1, *flatbuffers*::Offset<*flatbuffers*::Vector<uint16_t>> depth = 0)

inline *flatbuffers*::Offset<*DepthFrameFlatbuffer*> **CreateDepthFrameDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, int16_t sizeX = 0, int16_t sizeY = 0, uint16_t minDepth = 0, uint16_t maxDepth = 65535, uint16_t step = 1, const *std*::vector<uint16_t> *depth = nullptr)

inline *flatbuffers*::Offset<*DepthFrameFlatbuffer*> **CreateDepthFrame** (*flatbuffers*::FlatBufferBuilder &_fbb, const *DepthFrame* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*DepthFrameFlatbuffer* ***GetDepthFrame** (const void *buf)

inline const *dv*::*DepthFrameFlatbuffer* ***GetSizePrefixedDepthFrame** (const void *buf)

inline const char ***DepthFrameIdentifier** ()

inline bool **DepthFrameBufferHasIdentifier** (const void *buf)

inline bool **VerifyDepthFrameBuffer** (*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedDepthFrameBuffer** (*flatbuffers*::Verifier &verifier)

inline void **FinishDepthFrameBuffer** (*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*DepthFrameFlatbuffer*> root)

inline void **FinishSizePrefixedDepthFrameBuffer** (*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*DepthFrameFlatbuffer*> root)

inline *std*::unique_ptr<*DepthFrame*> **UnPackDepthFrame** (const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==** (const Event &lhs, const Event &rhs)

inline bool **operator==** (const *EventPacket* &lhs, const *EventPacket* &rhs)

inline const *flatbuffers*::TypeTable ***EventTypeTable** ()

inline const *flatbuffers*::TypeTable ***EventPacketTypeTable** ()

FLATBUFFERS_STRUCT_END (Event, 16)

inline *flatbuffers*::Offset<*EventPacketFlatbuffer*> **CreateEventPacket** (*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<const Event*>> elements = 0)

inline *flatbuffers*::Offset<*EventPacketFlatbuffer*> **CreateEventPacketDirect** (*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<Event> *elements = nullptr)

inline *flatbuffers*::Offset<*EventPacketFlatbuffer*> **CreateEventPacket** (*flatbuffers*::FlatBufferBuilder &_fbb, const *EventPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*EventPacketFlatbuffer* ***GetEventPacket** (const void *buf)

inline const *dv*::*EventPacketFlatbuffer* ***GetSizePrefixedEventPacket** (const void *buf)

inline const char *__EventPacketIdentifier__()

inline bool __EventPacketBufferHasIdentifier__(const void *buf)

inline bool __VerifyEventPacketBuffer__(*flatbuffers*::Verifier &verifier)

inline bool __VerifySizePrefixedEventPacketBuffer__(*flatbuffers*::Verifier &verifier)

inline void __FinishEventPacketBuffer__(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*EventPacketFlatbuffer*> root)

inline void __FinishSizePrefixedEventPacketBuffer__(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*EventPacketFlatbuffer*> root)

inline *std*::unique_ptr<*EventPacket*> __UnPackEventPacket__(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool __operator==__(const *Frame* &lhs, const *Frame* &rhs)

inline const *flatbuffers*::TypeTable *__FrameTypeTable__()

inline const *FrameFormat* (&__EnumValuesFrameFormat__())[32]

inline const char *const *__EnumNamesFrameFormat__()

inline const char *__EnumNameFrameFormat__(*FrameFormat* e)

inline const *FrameSource* (&__EnumValuesFrameSource__())[8]

inline const char *const *__EnumNamesFrameSource__()

inline const char *__EnumNameFrameSource__(*FrameSource* e)

inline *flatbuffers*::Offset<*FrameFlatbuffer*> __CreateFrame__(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, int64_t timestampStartOfFrame = 0, int64_t timestampEndOfFrame = 0, int64_t timestampStartOfExposure = 0, int64_t timestampEndOfExposure = 0, *FrameFormat* format = *FrameFormat*::*OPENCV_8U_C1*, int16_t sizeX = 0, int16_t sizeY = 0, int16_t positionX = 0, int16_t positionY = 0, *flatbuffers*::Offset<*flatbuffers*::Vector<uint8_t>> pixels = 0, int64_t exposure = 0, *FrameSource* source = *FrameSource*::*UNDEFINED*)

inline *flatbuffers*::Offset<*FrameFlatbuffer*> __CreateFrameDirect__(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, int64_t timestampStartOfFrame = 0, int64_t timestampEndOfFrame = 0, int64_t timestampStartOfExposure = 0, int64_t timestampEndOfExposure = 0, *FrameFormat* format = *FrameFormat*::*OPENCV_8U_C1*, int16_t sizeX = 0, int16_t sizeY = 0, int16_t positionX = 0, int16_t positionY = 0, const *std*::vector<uint8_t> *pixels = nullptr, int64_t exposure = 0, *FrameSource* source = *FrameSource*::*UNDEFINED*)

inline *flatbuffers*::Offset<*FrameFlatbuffer*> **CreateFrame** (*flatbuffers*::FlatBufferBuilder &_fbb, const *Frame* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *flatbuffers*::TypeTable ***FrameFormatTypeTable** ()

inline const *flatbuffers*::TypeTable ***FrameSourceTypeTable** ()

inline const *dv*::*FrameFlatbuffer* ***GetFrame** (const void *buf)

inline const *dv*::*FrameFlatbuffer* ***GetSizePrefixedFrame** (const void *buf)

inline const char ***FrameIdentifier** ()

inline bool **FrameBufferHasIdentifier** (const void *buf)

inline bool **VerifyFrameBuffer** (*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedFrameBuffer** (*flatbuffers*::Verifier &verifier)

inline void **FinishFrameBuffer** (*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*FrameFlatbuffer*> root)

inline void **FinishSizePrefixedFrameBuffer** (*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*FrameFlatbuffer*> root)

inline *std*::unique_ptr<*Frame*> **UnPackFrame** (const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==** (const Point3f &lhs, const Point3f &rhs)

inline bool **operator==** (const Point2f &lhs, const Point2f &rhs)

inline bool **operator==** (const Vec3f &lhs, const Vec3f &rhs)

inline bool **operator==** (const Vec2f &lhs, const Vec2f &rhs)

inline bool **operator==** (const Quaternion &lhs, const Quaternion &rhs)

inline const *flatbuffers*::TypeTable ***Point3fTypeTable** ()

inline const *flatbuffers*::TypeTable ***Point2fTypeTable** ()

inline const *flatbuffers*::TypeTable ***Vec3fTypeTable** ()

inline const *flatbuffers*::TypeTable ***Vec2fTypeTable** ()

inline const *flatbuffers*::TypeTable ***QuaternionTypeTable** ()

FLATBUFFERS_MANUALLY_ALIGNED_STRUCT (4) Point3f final

Structure representing absolute position of a 3D point.

Quaternion with Eigen compatible memory layout, should follow the Hamilton convention.

Structure representing a 2D vector.

Structure representing a 3D vector.

Structure representing absolute position of a 2D point.

FLATBUFFERS_STRUCT_END (Point3f, 12)

FLATBUFFERS_STRUCT_END (Point2f, 8)

FLATBUFFERS_STRUCT_END (Vec3f, 12)

FLATBUFFERS_STRUCT_END (Vec2f, 8)

FLATBUFFERS_STRUCT_END (Quaternion, 16)

inline bool **operator==**(const *IMU* &lhs, const *IMU* &rhs)

inline bool **operator==**(const *IMUPacket* &lhs, const *IMUPacket* &rhs)

inline const *flatbuffers*::TypeTable *__**IMUTypeTable**__()

inline const *flatbuffers*::TypeTable *__**IMUPacketTypeTable**__()

inline *flatbuffers*::Offset<*IMUFlatbuffer*> **CreateIMU**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, float temperature = 0.0f, float accelerometerX = 0.0f, float accelerometerY = 0.0f, float accelerometerZ = 0.0f, float gyroscopeX = 0.0f, float gyroscopeY = 0.0f, float gyroscopeZ = 0.0f, float magnetometerX = 0.0f, float magnetometerY = 0.0f, float magnetometerZ = 0.0f)

inline *flatbuffers*::Offset<*IMUFlatbuffer*> **CreateIMU**(*flatbuffers*::FlatBufferBuilder &_fbb, const *IMU* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*IMUPacketFlatbuffer*> **CreateIMUPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*IMUF* elements = 0)

inline *flatbuffers*::Offset<*IMUPacketFlatbuffer*> **CreateIMUPacketDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<*flatbuffers*::Offset<*IMUFlatbuffer*>> *elements = nullptr)

inline *flatbuffers*::Offset<*IMUPacketFlatbuffer*> **CreateIMUPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, const *IMUPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*IMUPacketFlatbuffer* *__**GetIMUPacket**__(const void *buf)

inline const *dv*::*IMUPacketFlatbuffer* *__**GetSizePrefixedIMUPacket**__(const void *buf)

inline const char *__**IMUPacketIdentifier**__()

inline bool **IMUPacketBufferHasIdentifier**(const void *buf)

inline bool **VerifyIMUPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedIMUPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishIMUPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*IMUPacketFlatbuffer*> root)

inline void **FinishSizePrefixedIMUPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*IMUPacketFlatbuffer*> root)

inline *std*::unique_ptr<*IMUPacket*> **UnPackIMUPacket**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==**(const *Observation* &lhs, const *Observation* &rhs)

inline bool **operator==**(const *Landmark* &lhs, const *Landmark* &rhs)

inline bool **operator==**(const *LandmarksPacket* &lhs, const *LandmarksPacket* &rhs)

inline const *flatbuffers*::TypeTable *__**ObservationTypeTable**__()

inline const *flatbuffers*::TypeTable *__**LandmarkTypeTable**__()

inline const *flatbuffers*::TypeTable *__**LandmarksPacketTypeTable**__()

inline *flatbuffers*::Offset<*ObservationFlatbuffer*> **CreateObservation**(*flatbuffers*::FlatBufferBuilder &_fbb, int32_t trackId = 0, int32_t cameraId = 0, *flatbuffers*::Offset<*flatbuffers*::String> cameraName = 0, int64_t timestamp = 0)

inline *flatbuffers*::Offset<*ObservationFlatbuffer*> **CreateObservationDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, int32_t trackId = 0, int32_t cameraId = 0, const char *cameraName = nullptr, int64_t timestamp = 0)

inline *flatbuffers*::Offset<*ObservationFlatbuffer*> **CreateObservation**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Observation* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*LandmarkFlatbuffer*> **CreateLandmark**(*flatbuffers*::FlatBufferBuilder &_fbb, const Point3f *pt = 0, int64_t id = 0, int64_t timestamp = 0, *flatbuffers*::Offset<*flatbuffers*::Vector<int8_t>> descriptor = 0, *flatbuffers*::Offset<*flatbuffers*::String> descriptorType = 0, *flatbuffers*::Offset<*flatbuffers*::Vector<float>> covariance = 0, *flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offset<*Observa*
observations = 0)

inline *flatbuffers*::Offset<*LandmarkFlatbuffer*> **CreateLandmarkDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, const Point3f *pt = 0, int64_t id = 0, int64_t timestamp = 0, const *std*::vector<int8_t> *descriptor = nullptr, const char *descriptorType = nullptr, const *std*::vector<float> *covariance = nullptr, const *std*::vector<*flatbuffers*::Offset<*ObservationFlatbuffer*> *observations = nullptr)

inline *flatbuffers*::Offset<*LandmarkFlatbuffer*> **CreateLandmark**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Landmark* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*LandmarksPacketFlatbuffer*> **CreateLandmarksPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffe* elements = 0, *flatbuffers*::Offset<*flatbuffers*::String> referenceFrame = 0)

inline *flatbuffers*::Offset<*LandmarksPacketFlatbuffer*> **CreateLandmarksPacketDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<*flatbuffers*::Offset<*Lan* *elements = nullptr, const char *referenceFrame = nullptr)

inline *flatbuffers*::Offset<*LandmarksPacketFlatbuffer*> **CreateLandmarksPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, const *LandmarksPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*LandmarksPacketFlatbuffer* ***GetLandmarksPacket**(const void *buf)

inline const *dv*::*LandmarksPacketFlatbuffer* ***GetSizePrefixedLandmarksPacket**(const void *buf)

inline const char ***LandmarksPacketIdentifier**()

inline bool **LandmarksPacketBufferHasIdentifier**(const void *buf)

inline bool **VerifyLandmarksPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedLandmarksPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishLandmarksPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*LandmarksPacketFlatbuffer*> root)

inline void **FinishSizePrefixedLandmarksPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*LandmarksPacketFlatbuffer*> root)

inline *std*::unique_ptr<*LandmarksPacket*> **UnPackLandmarksPacket**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==**(const *Pose* &lhs, const *Pose* &rhs)

inline const *flatbuffers*::TypeTable ***PoseTypeTable**()

inline *flatbuffers*::Offset<*PoseFlatbuffer*> **CreatePose**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, const Vec3f *translation = 0, const Quaternion *rotation = 0, *flatbuffers*::Offset<*flatbuffers*::String> referenceFrame = 0, *flatbuffers*::Offset<*flatbuffers*::String> targetFrame = 0)

inline *flatbuffers*::Offset<*PoseFlatbuffer*> **CreatePoseDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, const Vec3f *translation = 0, const Quaternion *rotation = 0, const char *referenceFrame = nullptr, const char *targetFrame = nullptr)

inline *flatbuffers*::Offset<*PoseFlatbuffer*> **CreatePose**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Pose* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*PoseFlatbuffer* ***GetPose**(const void *buf)

inline const *dv*::*PoseFlatbuffer* ***GetSizePrefixedPose**(const void *buf)

inline const char ***PoseIdentifier**()

inline bool **PoseBufferHasIdentifier**(const void *buf)

inline bool **VerifyPoseBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedPoseBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishPoseBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*PoseFlatbuffer*> root)

inline void **FinishSizePrefixedPoseBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*PoseFlatbuffer*> root)

inline *std*::unique_ptr<*Pose*> **UnPackPose**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==**(const *TimedKeyPoint* &lhs, const *TimedKeyPoint* &rhs)

inline bool **operator==**(const *TimedKeyPointPacket* &lhs, const *TimedKeyPointPacket* &rhs)

inline const *flatbuffers*::TypeTable ***TimedKeyPointTypeTable**()

inline const *flatbuffers*::TypeTable ***TimedKeyPointPacketTypeTable**()

inline *flatbuffers*::Offset<*TimedKeyPointFlatbuffer*> **CreateTimedKeyPoint**(*flatbuffers*::FlatBufferBuilder &_fbb, const Point2f *pt = 0, float size = 0.0f, float angle = 0.0f, float response = 0.0f, int32_t octave = 0, int32_t class_id = 0, int64_t timestamp = 0)

inline *flatbuffers*::Offset<*TimedKeyPointFlatbuffer*> **CreateTimedKeyPoint**(*flatbuffers*::FlatBufferBuilder &_fbb, const *TimedKeyPoint* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*TimedKeyPointPacketFlatbuffer*> **CreateTimedKeyPointPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector elements = 0)

inline *flatbuffers*::Offset<*TimedKeyPointPacketFlatbuffer*> **CreateTimedKeyPointPacketDirect**(*flatbuffers*::FlatBufferBuilde &_fbb, const *std*::vector<*flatbuffers*::C *elements = nullptr)

inline *flatbuffers*::Offset<*TimedKeyPointPacketFlatbuffer*> **CreateTimedKeyPointPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, const *TimedKeyPointPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*TimedKeyPointPacketFlatbuffer* ***GetTimedKeyPointPacket**(const void *buf)

inline const *dv*::*TimedKeyPointPacketFlatbuffer* ***GetSizePrefixedTimedKeyPointPacket**(const void *buf)

inline const char ***TimedKeyPointPacketIdentifier**()

inline bool **TimedKeyPointPacketBufferHasIdentifier**(const void *buf)

inline bool **VerifyTimedKeyPointPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedTimedKeyPointPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishTimedKeyPointPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*TimedKeyPointPacketFlatbuffer*> root)

inline void **FinishSizePrefixedTimedKeyPointPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*TimedKeyPointPacketFlatbuffer*> root)

inline *std*::unique_ptr<*TimedKeyPointPacket*> **UnPackTimedKeyPointPacket**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==**(const *Trigger* &lhs, const *Trigger* &rhs)

inline bool **operator==**(const *TriggerPacket* &lhs, const *TriggerPacket* &rhs)

inline const *flatbuffers*::TypeTable *__**TriggerTypeTable**()

inline const *flatbuffers*::TypeTable *__**TriggerPacketTypeTable**()

inline const *TriggerType* (&**EnumValuesTriggerType**())[10]

inline const char *const *__**EnumNamesTriggerType**()

inline const char *__**EnumNameTriggerType**(*TriggerType* e)

inline *flatbuffers*::Offset<*TriggerFlatbuffer*> **CreateTrigger**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t timestamp = 0, *TriggerType* type = *TriggerType*::*TIMESTAMP_RESET*)

inline *flatbuffers*::Offset<*TriggerFlatbuffer*> **CreateTrigger**(*flatbuffers*::FlatBufferBuilder &_fbb, const *Trigger* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*TriggerPacketFlatbuffer*> **CreateTriggerPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offs elements = 0)

inline *flatbuffers*::Offset<*TriggerPacketFlatbuffer*> **CreateTriggerPacketDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<*flatbuffers*::Offset<*TriggerFla *elements = nullptr)

inline *flatbuffers*::Offset<*TriggerPacketFlatbuffer*> **CreateTriggerPacket**(*flatbuffers*::FlatBufferBuilder &_fbb, const *TriggerPacket* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *flatbuffers*::TypeTable *__**TriggerTypeTypeTable**()

inline const *dv*::*TriggerPacketFlatbuffer* *__**GetTriggerPacket**(const void *buf)

inline const *dv*::*TriggerPacketFlatbuffer* *__**GetSizePrefixedTriggerPacket**(const void *buf)

inline const char *__**TriggerPacketIdentifier**()

inline bool **TriggerPacketBufferHasIdentifier**(const void *buf)

inline bool **VerifyTriggerPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedTriggerPacketBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishTriggerPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*TriggerPacketFlatbuffer*> root)

inline void **FinishSizePrefixedTriggerPacketBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*TriggerPacketFlatbuffer*> root)

inline *std*::unique_ptr<*TriggerPacket*> **UnPackTriggerPacket**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==**(const PacketHeader &lhs, const PacketHeader &rhs)

inline bool **operator==**(const *FileDataDefinition* &lhs, const *FileDataDefinition* &rhs)

inline bool **operator==**(const *FileDataTable* &lhs, const *FileDataTable* &rhs)

inline const *flatbuffers*::TypeTable ***PacketHeaderTypeTable**()

inline const *flatbuffers*::TypeTable ***FileDataDefinitionTypeTable**()

inline const *flatbuffers*::TypeTable ***FileDataTableTypeTable**()

FLATBUFFERS_STRUCT_END (PacketHeader, 8)

inline *flatbuffers*::Offset<*FileDataDefinitionFlatbuffer*> **CreateFileDataDefinition**(*flatbuffers*::FlatBufferBuilder &_fbb, int64_t ByteOffset = 0, const PacketHeader *PacketInfo = 0, int64_t NumElements = 0, int64_t TimestampStart = 0, int64_t TimestampEnd = 0)

inline *flatbuffers*::Offset<*FileDataDefinitionFlatbuffer*> **CreateFileDataDefinition**(*flatbuffers*::FlatBufferBuilder &_fbb, const *FileDataDefinition* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline *flatbuffers*::Offset<*FileDataTableFlatbuffer*> **CreateFileDataTable**(*flatbuffers*::FlatBufferBuilder &_fbb, *flatbuffers*::Offset<*flatbuffers*::Vector<*flatbuffers*::Offs Table = 0)

inline *flatbuffers*::Offset<*FileDataTableFlatbuffer*> **CreateFileDataTableDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, const *std*::vector<*flatbuffers*::Offset<*FileDataT *Table = nullptr)

inline *flatbuffers*::Offset<*FileDataTableFlatbuffer*> **CreateFileDataTable**(*flatbuffers*::FlatBufferBuilder &_fbb, const *FileDataTable* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *dv*::*FileDataTableFlatbuffer* ***GetFileDataTable**(const void *buf)

inline const *dv*::*FileDataTableFlatbuffer* ***GetSizePrefixedFileDataTable**(const void *buf)

inline const char ***FileDataTableIdentifier**()

inline bool **FileDataTableBufferHasIdentifier**(const void *buf)

inline bool **VerifyFileDataTableBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedFileDataTableBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishFileDataTableBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*FileDataTableFlatbuffer*> root)

inline void **FinishSizePrefixedFileDataTableBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*FileDataTableFlatbuffer*> root)

inline *std*::unique_ptr<*FileDataTable*> **UnPackFileDataTable**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

inline bool **operator==**(const *IOHeader* &lhs, const *IOHeader* &rhs)

inline const *flatbuffers*::TypeTable ***IOHeaderTypeTable**()

inline const *Constants* (&**EnumValuesConstants**())[1]

inline const char *const ***EnumNamesConstants**()

inline const char ***EnumNameConstants**(*Constants* e)

inline const *CompressionType* (&**EnumValuesCompressionType**())[5]

inline const char *const ***EnumNamesCompressionType**()

inline const char ***EnumNameCompressionType**(*CompressionType* e)

inline *flatbuffers*::Offset<*IOHeaderFlatbuffer*> **CreateIOHeader**(*flatbuffers*::FlatBufferBuilder &_fbb, *CompressionType* compression = *CompressionType*::*NONE*, int64_t dataTablePosition = -1, *flatbuffers*::Offset<*flatbuffers*::String> infoNode = 0)

inline *flatbuffers*::Offset<*IOHeaderFlatbuffer*> **CreateIOHeaderDirect**(*flatbuffers*::FlatBufferBuilder &_fbb, *CompressionType* compression = *CompressionType*::*NONE*, int64_t dataTablePosition = -1, const char *infoNode = nullptr)

inline *flatbuffers*::Offset<*IOHeaderFlatbuffer*> **CreateIOHeader**(*flatbuffers*::FlatBufferBuilder &_fbb, const *IOHeader* *_o, const *flatbuffers*::rehasher_function_t *_rehasher = nullptr)

inline const *flatbuffers*::TypeTable *__ConstantsTypeTable**()**

inline const *flatbuffers*::TypeTable *__CompressionTypeTypeTable**()**

inline const *dv*::*IOHeaderFlatbuffer* *__GetIOHeader**(**const void *buf**)**

inline const *dv*::*IOHeaderFlatbuffer* *__GetSizePrefixedIOHeader**(**const void *buf**)**

inline const char *__IOHeaderIdentifier**()**

inline bool **IOHeaderBufferHasIdentifier**(const void *buf)

inline bool **VerifyIOHeaderBuffer**(*flatbuffers*::Verifier &verifier)

inline bool **VerifySizePrefixedIOHeaderBuffer**(*flatbuffers*::Verifier &verifier)

inline void **FinishIOHeaderBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*IOHeaderFlatbuffer*> root)

inline void **FinishSizePrefixedIOHeaderBuffer**(*flatbuffers*::FlatBufferBuilder &fbb, *flatbuffers*::Offset<*dv*::*IOHeaderFlatbuffer*> root)

inline *std*::unique_ptr<*IOHeader*> **UnPackIOHeader**(const void *buf, const *flatbuffers*::resolver_function_t *res = nullptr)

Variables

static constexpr bool **DEBUG_ENABLED** = {true}

static constexpr *EventColor* **colorKeys**[4][4] = {{*EventColor*::*RED*, *EventColor*::*GREEN*, *EventColor*::*GREEN*, *EventColor*::*BLUE*}, {*EventColor*::*GREEN*, *EventColor*::*BLUE*, *EventColor*::*RED*, *EventColor*::*GREEN*}, {*EventColor*::*GREEN*, *EventColor*::*RED*, *EventColor*::*BLUE*, *EventColor*::*GREEN*}, {*EventColor*::*BLUE*, *EventColor*::*GREEN*, *EventColor*::*GREEN*, *EventColor*::*RED*},}

Address to Color mapping for events based on Bayer filter.

static constexpr int **VERSION_MAJOR** = {1}

static constexpr int **VERSION_MINOR** = {7}

static constexpr int **VERSION_PATCH** = {9}

static constexpr int **VERSION** = {((1 * 10000) + (7 * 100) + 9)}

static constexpr *std*::string_view **NAME_STRING** = {"dv-processing"}

static constexpr *std*::string_view **VERSION_STRING** = {"1.7.9"}

namespace **dv**

namespace **camera**

Enums

enum **DistortionModel**

Values:

enumerator **None**

enumerator **RadTan**

enumerator **Equidistant**

Functions

static *DistortionModel* **stringToDistortionModel**(const *std*::string_view model)

Convert a string into the Enum of the DistortionModel

Parameters
model –

Returns
the enum corresponding to the string

static *std*::string **distortionModelToString**(const *DistortionModel* &model)

Convert a DistortionModel Enum into a string

Parameters
model –

Returns
the string that represent the Distortion model

namespace **calibrations**

namespace **internal**

Variables

static constexpr *std*::string_view **NoneModelString** = {"none"}

static constexpr *std*::string_view **RadialTangentialModelString** = {"radialTangential"}

static constexpr *std*::string_view **EquidistantModelString** = {"equidistant"}

namespace **cluster**

namespace **mean_shift**

Typedefs

template<typename **TYPE**, int32_t **ROWS** = Eigen::Dynamic, int32_t **COLUMNS** = Eigen::Dynamic>

using **MeanShiftRowMajorMatrixXX** = *MeanShiftEigenMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, Eigen::RowMajor>

Convenience alias for n-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **ROWS** = Eigen::Dynamic, int32_t **COLUMNS** = Eigen::Dynamic>

using **MeanShiftColMajorMatrixXX** = *MeanShiftEigenMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, Eigen::ColMajor>

Convenience alias for n-dimensional data in column-major sample order of arbitrary dimensions and number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftRowMajorMatrixX1** = *MeanShiftRowMajorMatrixXX*<*TYPE*, *SAMPLES*, 1>

Convenience alias for 1-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftRowMajorMatrixX2** = *MeanShiftRowMajorMatrixXX*<*TYPE*, *SAMPLES*, 2>

Convenience alias for3-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftRowMajorMatrixX3** = *MeanShiftRowMajorMatrixXX*<*TYPE*, *SAMPLES*, 3>

Convenience alias for 3-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftRowMajorMatrixX4** = *MeanShiftRowMajorMatrixXX*<*TYPE*, *SAMPLES*, 4>

Convenience alias for 4-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftColMajorMatrix1X** = *MeanShiftColMajorMatrixXX*<*TYPE*, 1, *SAMPLES*>

Convenience alias for 1-dimensional data in column-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftColMajorMatrix2X** = *MeanShiftColMajorMatrixXX*<*TYPE*, 2, *SAMPLES*>

Convenience alias for 2-dimensional data in column-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftColMajorMatrix3X** = *MeanShiftColMajorMatrixXX*<*TYPE*, 3, *SAMPLES*>

Convenience alias for 3-dimensional data in column-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **MeanShiftColMajorMatrix4X** = *MeanShiftColMajorMatrixXX*<*TYPE*, 4, *SAMPLES*>

Convenience alias for 4-dimensional data in column-major sample order of arbitrary number of samples

namespace **kernel**

namespace **concepts**

Typedefs

template<class **T**>

using **iterable_element_type** = typename *std*::remove_reference_t<decltype(*(*std*::declval<*T*>().begin()))>

Variables

template<typename **T**>

constexpr bool **is_eigen_type** = *internal*::*is_eigen_impl*<*T*>::value

template<typename **Needle**, typename ...**Haystack**>

constexpr bool **is_type_one_of** = *std*::disjunction_v<*std*::is_same<*Needle*, *Haystack*>...>

namespace **internal**

namespace **containers**

namespace **kd_tree**

Typedefs

template<typename **TYPE**, int32_t **ROWS** = Eigen::Dynamic, int32_t **COLUMNS** = Eigen::Dynamic>

using **KDTreeRowMajorXX** = *KDTreeMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, Eigen::RowMajor>

Convenience alias for n-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **ROWS** = Eigen::Dynamic, int32_t **COLUMNS** = Eigen::Dynamic>

using **KDTreeColMajorXX** = *KDTreeMatrixAdaptor*<*TYPE*, *ROWS*, *COLUMNS*, Eigen::ColMajor>

Convenience alias for n-dimensional data in column-major sample order of arbitrary dimensions and number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeRowMajorX1** = *KDTreeRowMajorXX*<*TYPE*, *SAMPLES*, 1>

Convenience alias for 1-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeRowMajorX2** = *KDTreeRowMajorXX*<*TYPE*, *SAMPLES*, 2>

Convenience alias for 2-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeRowMajorX3** = *KDTreeRowMajorXX*<*TYPE*, *SAMPLES*, 3>

Convenience alias for 3-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeRowMajorX4** = *KDTreeRowMajorXX*<*TYPE*, *SAMPLES*, 4>

Convenience alias for 4-dimensional data in row-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeColMajor1X** = *KDTreeColMajorXX*<*TYPE*, 1, *SAMPLES*>

Convenience alias for 1-dimensional data in column-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeColMajor2X** = *KDTreeColMajorXX*<*TYPE*, 2, *SAMPLES*>

Convenience alias for 2-dimensional data in column-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeColMajor3X** = *KDTreeColMajorXX*<*TYPE*, 3, *SAMPLES*>

Convenience alias for 3-dimensional data in column-major sample order of arbitrary number of samples

template<typename **TYPE**, int32_t **SAMPLES** = Eigen::Dynamic>

using **KDTreeColMajor4X** = *KDTreeColMajorXX*<*TYPE*, 4, *SAMPLES*>

Convenience alias for 5-dimensional data in column-major sample order of arbitrary number of samples

namespace **data**

Functions

inline *std*::vector<cv::KeyPoint> **fromTimedKeyPoints**(const *dv*::*cvector*<*dv*::*TimedKeyPoint*> &points)

Convert *TimedKeyPoint* vector into cv::KeyPoint vector.

Parameters
points – KeyPoints to be converted.

Returns
A vector of cv::KeyPoint.

inline *std*::vector<cv::Point2f> **convertToCvPoints**(const *dv*::*cvector*<*dv*::*TimedKeyPoint*> &points)

Convert *TimedKeyPoint* vector into cv::Point2f vector.

Parameters
points – KeyPoints to be converted.

Returns
A vector of cv::Point2f.

inline *dv*::*cvector*<*dv*::*TimedKeyPoint*> **fromCvKeypoints**(const *std*::vector<cv::KeyPoint> &points, const int64_t defaultTime = 0)

Create a vector of `cv::KeyPoint` from a given vector of *`dv::TimedKeyPoint`*.

Parameters

- **points** – cv::KeyPoint vector to be converted.
- **defaultTime** – Timestamp in microseconds to be assigned to all new TimedKeyPoints.

Returns
A vector of TimedKeyPoints.

inline cv::Mat **depthFrameMap**(*dv*::*DepthFrame* &frame)

Map a depth frame into an OpenCV Mat, no data copies are performed. The resulting cv::Mat will point to the same underlying data.

This function does not affect any data underlying, the const qualifier is not set since cv::Mat can't be const.

Parameters
frame – *Frame* to be mapped.

Returns
Mapped depth frame in `cv::Mat` with data type of `CV_16UC1`.

inline cv::Mat **depthFrameInMeters**(*dv*::*DepthFrame* &frame)

Converts the given depth frame into an OpenCV matrix containing depth values in meters.

Resulting `cv::Mat` will be of floating type and will apply conversion from millimeters to meters. Depth value of 0.0f should be considered invalid.

This function will copy and scale all values into meters.

Parameters
frame – Depth frame to be converted.

Returns
A `cv::Mat` containing scaled depth values in meters.

inline *dv*::*DepthFrame* **depthFrameFromCvMat**(const cv::Mat &depthImage)

Converts the given OpenCV matrix with depth values to *DepthFrame*.

`cv::Mat` can contain single-channel floating point containing depth values in meters or single-channel 16-bit unsigned integer values in millimeters. Zero should be used for invalid values.

This function will copy and scale all values into millimeter 16-bit integer representation.

Parameters
depthImage – `cv::Mat` containing the depth values.

Returns
Depth frame containing depth values in 16-bit unsigned integer values representing distance in millimeters.

template<*std*::floating_point **Scalar** = float>
inline *dv*::*kinematics*::*Transformation*<*Scalar*> **transformFromPose**(const *dv*::*Pose* &pose)

Convert a pose message into a transformation.

Parameters
pose – Input pose to be converted.

Returns
Transformation representing the pose.

template<*std*::floating_point **Scalar** = float>
inline *dv*::*Pose* **poseFromTransformation**(const *dv*::*kinematics*::*Transformation*<*Scalar*> &transform)

Convert a transformation into a pose message.

Parameters
transform – Input transform.

Returns
Pose message representing the transform.

namespace **generate**

Functions

inline cv::Mat **sampleImage**(const cv::Size &resolution)

Generate a sample image (single channel 8-bit unsigned integer) containing a few gray rectangles in a black background.

Parameters
resolution – Resolution of the output image.

Returns
Generated image.

inline *dv*::*EventStore* **eventLine**(const int64_t timestamp, const cv::Point &a, const cv::Point &b, size_t steps = 0)

Generate events along a line between two given end-points.

Parameters

- **timestamp** – Fixed timestamp assigned for all events.
- **a** – Starting point.
- **b** – Ending point.

- **steps** – Number of events generated for the line. If zero is provided, the function uses euclidean distance between the points.

Returns
A batch of event along the line.

inline *dv*::*EventStore* **eventRectangle**(const int64_t timestamp, const cv::Point &tl, const cv::Point &br)

Generate events along a rectangle edges between two given top-left and bottom right points.

Parameters

- **timestamp** – Fixed timestamp assigned for all events.
- **tl** – Top left coordinate of the rectangle.
- **br** – Bottom right coordinate of the rectangle.

Returns
Event batch containing events at the edges of a given rectangle.

inline *dv*::*EventStore* **eventTestSet**(const int64_t timestamp, const cv::Size &resolution)

Generate an event test set that contains event for a few intersecting rectangle edges.

Parameters

- **timestamp** – Fixed timestamp assigned for all events.
- **resolution** – Expected resolution limits for the events.

Returns
Generated event batch.

inline *dv*::*EventStore* **uniformlyDistributedEvents**(const int64_t timestamp, const cv::Size &resolution, const size_t count, const uint64_t seed = 0)

Generate a batch of uniformly distributed set of event within the given resolution.

Parameters

- **timestamp** – Fixed timestamp assigned for all events.
- **resolution** – Resolution limits.
- **count** – Number of events.
- **seed** – Seed for the RNG.

Returns
Generated event batch.

inline *dv*::*EventStore* **normallyDistributedEvents**(const int64_t timestamp, const *dv*::Point2f &center, const *dv*::Point2f &stddev, const size_t count, const uint64_t seed = 0)

Generate events normally distributed around a given center coordinates with given standard deviation.

Parameters

- **timestamp** – Timestamp to be assigned to the generated events
- **center** – Center coordinates
- **stddev** – Standard deviation for each of the axes
- **count** – Number of events to generate
- **seed** – Seed for the RNG

Returns
Set of normally distributed events

inline *dv*::*EventStore* **uniformEventsWithinTimeRange**(const int64_t startTime, const *dv*::*Duration* duration, const cv::Size &resolution, const int64_t count, const uint64_t seed = 0)

Generate a batch of uniformly distributed (in pixel-space) randomly generated events. The timestamps are generated by monotonically increasing the timestamp within the time duration.

Parameters

- **startTime** – Start timestamp in microseconds.
- **duration** – Duration of the generated data.
- **resolution** – Pixel space resolution.
- **count** – Number of output events.
- **seed** – Seed for the RNG.

Returns
Generated event batch.

inline cv::Mat **dvLogo**(const cv::Size &size, const bool colored = true, const cv::Scalar &bgColor = *dv*::*visualization*::*colors*::*white*, const cv::Scalar &pColor = *dv*::*visualization*::*colors*::*iniBlue*, const cv::Scalar &nColor = *dv*::*visualization*::*colors*::*darkGrey*)

Generate a DV logo using simple drawing methods. Generates in color or grayscale.

Parameters

- **size** – Output dimensions of the drawing
- **colored** – Colored output (CV_8UC3) if true, or grayscale (CV_8UC1) otherwise.

Returns
Image containing DV logo.

inline *dv*::*EventStore* **imageToEvents**(const int64_t timestamp, const cv::Mat &image, const uint8_t positive, const uint8_t negative)

Convert an image into event by matching pixel intensities. The algorithm will match all pixel values available in the and match against positive and negative pixel intensity values, according events are going to be added into the output event store. Other pixel intensity values are ignored.

Parameters

- **image** – Input image for conversion
- **positive** – Pixel brightness intensity value to consider the pixel to generate a positive polarity event.
- **negative** – Pixel brightness intensity value to consider the pixel to generate a negative polarity event.

Returns
Generated events.

inline *dv*::*EventStore* **dvLogoAsEvents**(const int64_t timestamp, const cv::Size &resolution)

Generate a DV logo using simple drawing methods. Generates negative polarity events on the pixels where logo has dark pixels and positive polarity events where pixels have brighter events.

Parameters

- **timestamp** – Timestamp assigned to each generated event.
- **resolution** – Resolution of the events.

Returns
Events that can be accumulated / visualized to generate a logo of DV.

inline *dv*::*IMU* **levelImuMeasurement**(const int64_t timestamp)

Generate an *IMU* measurement that measures a camera being on a stable and level surface. All measurement values are going to be zero, except for Y axis of accelerometer, it will measure -1.0G.

Parameters
timestamp – Timestamp to be assigned to the measurement.

Returns
Generated *IMU* measurement.

inline *dv*::*IMU* **addNoiseToImu**(const *dv*::*IMU* &measurement, const float accelerometerStddev, const float gyroscopeStddev, const uint64_t seed = 0)

Apply noise to imu measurements (accelerometer and gyroscope). The noise is modelled as a normal distribution with 0 mean and given standard deviation. The modelled noise is added to the given measurement and return a new *dv::IMU* structure with added noise.

Parameters

- **measurement** – *IMU* measurement to add noise to.
- **accelerometerStddev** – Accelerometer noise standard deviation.
- **gyroscopeStddev** – Gyroscope noise standard deviation.
- **seed** – Seed for the RNG.

Returns
Generated measurement with added noise.

inline *dv*::*IMU* **levelImuWithNoise**(const int64_t timestamp, const float accelerometerStddev = 0.1f, const float gyroscopeStddev = 0.01f, const uint64_t seed = 0)

Generate an *IMU* measurement that measures a camera being on a stable and level surface with additional measurement noise. The noise is modelled as a normal distribution with 0 mean and given standard deviation.

Parameters

- **timestamp** – Timestamp to be assigned to the measurement.
- **accelerometerStddev** – Accelerometer noise standard deviation.
- **gyroscopeStddev** – Gyroscope noise standard deviation.
- **seed** – Seed for the RNG.

Returns
Generated *IMU* measurement.

namespace **depth**

Functions

inline *std*::shared_ptr<cv::StereoMatcher> **defaultStereoMatcher** ( )

Create a reasonable default stereo matcher, tailored for low texture images (that are generated by accumulating events) and for faster execution.

The method creates an instance of cv::StereoSGBM with following parameter values:

- minDisparity = 0
- numDisparities = 48
- blockSize = 11 : highest recommended block size, small block sizes generate noise in low texture)
- P1 = 8 * (blockSize ^ 2)
- P2 = 32 * (blockSize ^ 2) : P1 and P2 are calculated using recommended equations
- disp12MaxDiff = 0 : disparity is also calculated on right-left image pair, filter out any disparities that do not agree. This enables strong noise filtering (there can be a lot of noise due to low texture)
- preFilterCap = cv::StereoBM::PREFILTER_NORMALIZED_RESPONSE : disable Sobel filter preprocessing
- uniquenessRatio = 15 : this is also an aggressive value for a noise filter
- speckleWindowSize = 240 : this is also an aggressive value for a speckle noise filter
- speckleRange = 1 : this is also an aggressive value for a speckle noise filter
- mode = cv::StereoSGBM::MODE_SGBM_3WAY : Fastest disparity calculation mode

Returns
Stereo semi global block matching algorithm with reasonable defaults for low texture images.

namespace **exceptions**

Typedefs

using **DirectoryError** = *Exception_*<*info*::*DirectoryError*>

using **DirectoryNotFound** = *Exception_*<*info*::*DirectoryNotFound*, *DirectoryError*>

using **FileError** = *Exception_*<*info*::*FileError*>

using **FileOpenError** = *Exception_*<*info*::*FileOpenError*, *FileError*>

using **FileReadError** = *Exception_*<*info*::*FileReadError*, *FileError*>

using **FileWriteError** = *Exception_*<*info*::*FileWriteError*, *FileError*>

using **FileNotFound** = *Exception_*<*info*::*FileNotFound*, *FileError*>

using **AedatFileError** = *Exception_*<*info*::*AedatFileError*, *FileError*>

using **AedatVersionError** = *Exception_*<*info*::*AedatVersionError*, *AedatFileError*>

using **AedatFileParseError** = *Exception_*<*info*::*AedatFileParseError*, *AedatFileError*>

using **EndOfFile** = *Exception_*<*info*::*EndOfFile*>

using **RuntimeError** = *Exception_*<*info*::*RuntimeError*>

using **BadAlloc** = *Exception_*<*info*::*BadAlloc*>

using **OutOfRange** = *Exception_*<*info*::*OutOfRange*>

using **LengthError** = *Exception_*<*info*::*LengthError*>

template<class **TYPE**>

using **InvalidArgument** = *Exception_*<*info*::*InvalidArgument*<*TYPE*>>

using **NullPointer** = *Exception_*<*info*::*NullPointer*>

using **IOError** = *Exception_*<*info*::*IOError*>

using **InputError** = *Exception_*<*info*::*InputError*, *IOError*>

using **OutputError** = *Exception_*<*info*::*OutputError*, *IOError*>

using **TypeError** = *Exception_*<*info*::*TypeError*>

namespace **info**

namespace **internal**

Functions

template<*HasCustomExceptionFormatter* **T**>
std::string **format**(const typename *T*::Info &info)

namespace **features**

Typedefs

using **ImagePyrFeatureDetector** = *FeatureDetector*<*dv*::*features*::*ImagePyramid*, cv::Feature2D>

using **ImageFeatureDetector** = *FeatureDetector*<*dv*::*Frame*, cv::Feature2D>

using **EventFeatureBlobDetector** = *FeatureDetector*<*dv*::*EventStore*, *EventBlobDetector*>

namespace **internal**

This class implement the Arc* corner detector presented in the following paper: https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/277131/RAL2018-camera-ready.pdf

Template Parameters

- **radius1** – radius of the first circle on which the timestamps are checked for corner-ness
- **radius2** – radius of the second circle on which the timestamps are checked for corner-ness

namespace **imgproc**

Functions

template<typename **T**>
inline auto **cvMatStepToEigenStride**(const cv::MatStep &step)

Conversion from cv::MatStep to Eigen::Stride

cv::MatStep stores steps in units of bytes, as the underlying matrix is always stored in uint8_t arrays, which are then interpreted at run-time based on the type (e.g. CV_8U). Contrary to this, Eigen stores matrices in arrays of a type that is determined at compile-time based on a template argument, and therefore stores its strides in units of pointer increments. The conversion between the two can be computed by dividing by or multiplying with sizeof(T).

Template Parameters
T – the type of the scalars stored in the matrices

Parameters
step – the step (stride) in the matrix in units of bytes

Returns
the corresponding Eigen::Stride for the cv::MatStep value provided

template<typename **T**>
inline auto **cvMatToEigenMap**(const cv::Mat &mat)

Maps an Eigen::Map onto a cv::Mat object. This provides a view to the internal storage of the cv::Mat, it doesn't copy any data.

Template Parameters
T – the type of the scalars stored in the matrices

Parameters
mat – the cv::Mat onto which an Eigen::Map should be mapped

Returns
the view into the cv::Mat via an Eigen::Map object

template<typename **T**>

inline auto **cvMatToEigenMap**(cv::Mat &mat)

Maps an Eigen::Map onto a cv::Mat object. This provides a view to the internal storage of the cv::Mat, it doesn't copy any data.

Template Parameters
T – the type of the scalars stored in the matrices

Parameters
mat – the cv::Mat onto which an Eigen::Map should be mapped

Returns
the view into the cv::Mat via an Eigen::Map object

template<typename **T**>
inline auto **L1Distance**(const Eigen::Block<*T*, Eigen::Dynamic, Eigen::Dynamic> &patch1, const Eigen::Block<*T*, Eigen::Dynamic, Eigen::Dynamic> &patch2)

Computes the L1 distance between two blocks (patches) of eigen matrices.

Template Parameters
T – The type of the underlying matrix

Parameters

- **patch1** – the first patch
- **patch2** – the second patch

Returns
the L1 distance between the two patches

template<typename **T**, int32_t **MAP_OPTIONS**, typename **STRIDE**>
inline auto **L1Distance**(const Eigen::Map<*T*, *MAP_OPTIONS*, *STRIDE*> &m1, const Eigen::Map<*T*, *MAP_OPTIONS*, *STRIDE*> &m2)

Computes the L1 distance between two matrices

See also:

Eigen::Map::MapOptions

Template Parameters

- **T** – The type of the underlying matrix
- **MAP_OPTIONS** – The options for the underlying matrix.
- **STRIDE** – The stride of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the L1 distance between the two matrices

template<typename **T**>
inline auto **L1Distance**(const Eigen::Matrix<*T*, Eigen::Dynamic, Eigen::Dynamic> &m1, const Eigen::Matrix<*T*, Eigen::Dynamic, Eigen::Dynamic> &m2)

Computes the L1 distance between two matrices

Template Parameters
T – The type of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the L1 distance between the two matrices

inline auto **L1Distance**(const cv::Mat &m1, const cv::Mat &m2)

Computes the L1 distance between two matrices

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the L1 distance between the two matrices

template<typename **T**>
inline auto **pearsonCorrelation**(const Eigen::Block<*T*, Eigen::Dynamic, Eigen::Dynamic> &patch1, const Eigen::Block<*T*, Eigen::Dynamic, Eigen::Dynamic> &patch2)

Computes the Pearson Correlation between two blocks (patches) of eigen matrices.

Template Parameters
T – The type of the underlying matrix

Parameters

- **patch1** – the first patch
- **patch2** – the second patch

Returns
the Pearson Correlation between the two patches

template<typename **T**, int32_t **MAP_OPTIONS**, typename **STRIDE**>
inline auto **pearsonCorrelation**(const Eigen::Map<*T*, *MAP_OPTIONS*, *STRIDE*> &m1, const Eigen::Map<*T*, *MAP_OPTIONS*, *STRIDE*> &m2)

Computes the Pearson Correlation between two matrices

See also:

Eigen::Map::MapOptions

Template Parameters

- **T** – The type of the underlying matrix
- **MAP_OPTIONS** – The options for the underlying matrix.
- **STRIDE** – The stride of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the Pearson Correlation between the two matrices

template<typename **T**>

inline auto **pearsonCorrelation**(const Eigen::Matrix<*T*, Eigen::Dynamic, Eigen::Dynamic> &m1, const Eigen::Matrix<*T*, Eigen::Dynamic, Eigen::Dynamic> &m2)

Computes the Pearson Correlation between two matrices

Template Parameters
T – The type of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the Pearson Correlation between the two matrices

inline auto **pearsonCorrelation**(const cv::Mat &m1, const cv::Mat &m2)

Computes the Pearson Correlation between two matrices

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the Pearson Correlation between the two matrices

template<typename **T**>
inline auto **cosineDistance**(const Eigen::Block<*T*, Eigen::Dynamic, Eigen::Dynamic> &patch1, const Eigen::Block<*T*, Eigen::Dynamic, Eigen::Dynamic> &patch2)

Computes the Cosine Distance between two blocks (patches) of eigen matrices.

Template Parameters
T – The type of the underlying matrix

Parameters

- **patch1** – the first patch
- **patch2** – the second patch

Returns
the Cosine Distance between the two patches

template<typename **T**>
inline auto **cosineDistance**(const Eigen::Matrix<*T*, Eigen::Dynamic, Eigen::Dynamic> &m1, const Eigen::Matrix<*T*, Eigen::Dynamic, Eigen::Dynamic> &m2)

Computes the Cosine Distance between two matrices

Template Parameters
T – The type of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the Cosine Distance between the two matrices

template<typename **T**, int32_t **MAP_OPTIONS**, typename **STRIDE**>

inline auto **cosineDistance**(const Eigen::Map<*T*, *MAP_OPTIONS*, *STRIDE*> &m1, const Eigen::Map<*T*, *MAP_OPTIONS*, *STRIDE*> &m2)

Computes the Cosine Distance between two matrices

See also:

Eigen::Map::MapOptions

Template Parameters

- **T** – The type of the underlying matrix
- **MAP_OPTIONS** – The options for the underlying matrix.
- **STRIDE** – The stride of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the Cosine Distance between the two matrices

inline auto **cosineDistance**(const cv::Mat &m1, const cv::Mat &m2)

Computes the Cosine Distance between two matrices

Template Parameters
T – The type of the underlying matrix

Parameters

- **m1** – the first matrix
- **m2** – the second matrix

Returns
the Cosine Distance between the two matrices

namespace **imu**

namespace **io**

Typedefs

using **DataReadVariant** = *std*::variant<*dv*::*EventStore*, *dv*::*Frame*, *dv*::*cvector*<*dv*::*IMU*>, *dv*::*cvector*<*dv*::*Trigger*>, *DataReadHandler*::*OutputFlag*>

Enums

enum class **ModeFlags** : uint8_t

Values:

enumerator **READ**

enumerator **WRITE**

enum class **WriteFlags** : uint8_t

Values:

enumerator **NONE**

enumerator **TRUNCATE**

enumerator **APPEND**

enum class **SeekFlags** : int

Values:

enumerator **START**

enumerator **CURRENT**

enumerator **END**

Functions

static inline *std*::vector<*std*::string> **discoverDevices**()

Retrieve a list of connected cameras. The list will contain camera names, which are supported for `dv::io::CameraCapture` class.

Returns
A list of currently connected camera names.

inline *ModeFlags* **operator|**(const *ModeFlags* lhs, const *ModeFlags* rhs)

inline *ModeFlags* &**operator|=**(*ModeFlags* &lhs, const *ModeFlags* rhs)

inline bool **operator&**(const *ModeFlags* lhs, const *ModeFlags* rhs)

inline *WriteFlags* **operator|**(const *WriteFlags* lhs, const *WriteFlags* rhs)

inline *WriteFlags* &**operator|=**(*WriteFlags* &lhs, const *WriteFlags* rhs)

inline bool **operator&**(const *WriteFlags* lhs, const *WriteFlags* rhs)

namespace **compression**

Functions

static *std*::unique_ptr<*CompressionSupport*> **createCompressionSupport**(const *CompressionType* type)

static *std*::unique_ptr<*DecompressionSupport*> **createDecompressionSupport**(const *CompressionType* type)

namespace **encrypt**

Functions

inline asioSSL::context **createEncryptionContext**(asioSSL::context::method method, const *std*::filesystem::path &certificateChain, const *std*::filesystem::path &privateKey, const *std*::optional<*std*::filesystem::path> &CAFile = *std*::nullopt)

Create an encryption context.

Parameters

- **method** – Encryption mode.
- **certificateChain** – Path to certificate chain.
- **privateKey** – Path to a private key.
- **CAFile** – Path to CAFile, if a std::nullopt is provided, peer verification is disabled. Can be an empty path, in that case the context will use CA from default locations, the peers will be verified.

Returns

Encryption context.

inline asioSSL::context **defaultEncryptionServer**(const *std*::filesystem::path &certificateChain, const *std*::filesystem::path &privateKey, const *std*::filesystem::path &CAFile)

Create an encryption server context with default configuration: TLSv1.2 encryption algorithm, provided certificate chain, server private key and certificate authority CAFile which is used to verify client certificate.

Parameters

- **certificateChain** – Server certificate chain.
- **privateKey** – Server private key.
- **CAFile** – CAFile for client verification.

Returns

SSL context that can be used for encrypted network connections.

inline asioSSL::context **defaultEncryptionClient**(const *std*::filesystem::path &certificateChain, const *std*::filesystem::path &privateKey)

Create an encrypted client context with default configuration: TLSv1.2 encryption algorithm, provided client certificate chain and client private key. Server is always considered trusted and server certificate is not verified, the server will verify the client and can reject the connection during handshake if certificate verification fails.

Parameters

- **certificateChain** – Client certificate chain.

- **privateKey** – Client private key.

Returns
SSL context that can be used with encrypted network connections.

namespace **internal**

Functions

static inline *std*::string **chipIDToName**(const int16_t chipID)

Get a text representation of a chipID integer.

Parameters
chipID – Chip ID integer value.

Returns
Chip name string.

static inline *std*::string **getDiscoveredCameraName**(const caer_device_discovery_result &discovery)

Get a camera name from a discovered device structure.

Parameters
discovery – Discovered device.

Returns
A string with device name, that is used in the library to identify a unique device.

static inline *std*::string **imuModelString**(const caer_imu_types imuType)

Get *IMU* model name from a *IMU* type identifier.

Parameters
imuType – *IMU* type.

Returns
IMU model string.

namespace **network**

Typedefs

using **asioUNIX** = asioLocal::stream_protocol

using **asioTCP** = asioIP::tcp

namespace **support**

Typedefs

using **TypeResolver** = *dv*::*std_function_exact*<const *dv*::*types*::*Type**(const uint32_t)>

using **VariantValueOwning** = *std*::variant<bool, int32_t, int64_t, float, double, *std*::string>

Functions

static inline const *dv*::*types*::*Type* ***defaultTypeResolver**(const uint32_t typeId)

template<class **PacketType**>
inline *std*::shared_ptr<*dv*::*types*::*TypedObject*> **packetToObject**(*PacketType* &&packet, const *TypeResolver* &resolver = *defaultTypeResolver*)

Variables

static constexpr *std*::string_view **AEDAT4_FILE_EXTENSION** = {".aedat4"}

static constexpr *std*::string_view **AEDAT4_HEADER_VERSION** = {"#!AER-DAT4.0\r\n"}

namespace **kinematics**

Typedefs

typedef *LinearTransformer*<float> **LinearTransformerf**

LinearTransformer using single precision float operations

typedef *LinearTransformer*<double> **LinearTransformerd**

LinearTransformer using double precision float operations

typedef *Transformation*<float> **Transformationf**

Transformation using single precision float operations

typedef *Transformation*<double> **Transformationd**

Transformation using double precision float operations

namespace **measurements**

namespace **noise**

namespace **optimization**

namespace **packets**

Enums

enum class **Timestamp**

Values:

enumerator **START**

enumerator **END**

Functions

template<class **ElementType**>
inline int64_t **getTimestamp**(const *ElementType* &element)

Template method that retrieves timestamp from a Timestamped structure.

Template Parameters
ElementType – Type of the element

Parameters
element – Instance of the element

Returns
Timestamp of this element

template<class **PacketType**>
inline bool **isPacketEmpty**(const *PacketType* &packet)

Check if a packet is empty.

Template Parameters
PacketType –

Parameters
packet –

Returns
True if the given packet is empty, false otherwise.

template<class **PacketType**>
inline size_t **getPacketSize**(const *PacketType* &packet)

Get packet size. This utility template method can be used to generically get size of a EventStore, data packet or any container satisfying the iterable concept.

Template Parameters
PacketType –

Parameters
packet –

Returns
Size of the given packet

template<class **PacketType**>
inline auto **getPacketBegin**(const *PacketType* &packet)

Generic getter of a begin iterator of a packet.

Template Parameters
PacketType –

Parameters
packet –

Returns

template<class **PacketType**>
inline auto **getPacketEnd**(const *PacketType* &packet)

Generic getter of an end iterator of a packet.

Template Parameters
PacketType –

Parameters
packet –

Returns

template<*Timestamp* **startTime**, class **PacketType**>
inline int64_t **getPacketTimestamp**(const *PacketType* &packet)

Retrieve packet start or end timestamp using template generation.

Template Parameters

- **startTime** – Use enum to select whether you want start or end timestamp.
- **PacketType** – Packet type, inferred from argument type.

Parameters
packet – Non-empty data packet.

Throws
`InvalidArgument` – exception is thrown if the packet is empty.

Returns
Timestamp of the first or last element in the packet.

template<class **PacketType**>
inline *dv*::*TimeWindow* **getPacketTimeWindow**(const *PacketType* &packet)

Get time window for a given packet.

Template Parameters
PacketType –

Parameters
packet – Non-empty data packet.

Throws
`InvalidArgument` – exception is thrown if the packet is empty.

Returns
Time window with start and end timestamps of this packet.

namespace **types**

Typedefs

using **PackFuncPtr** = *std*::add_pointer_t<uint32_t(void *toFlatBufferBuilder, const void *fromObject)>

using **UnpackFuncPtr** = *std*::add_pointer_t<void(void *toObject, const void *fromFlatBuffer)>

using **ConstructPtr** = *std*::add_pointer_t<void*(const size_t sizeOfObject)>

using **DestructPtr** = *std*::add_pointer_t<void(void *object)>

using **TimeElementExtractorPtr** = *std*::add_pointer_t<void(const void *object, *TimeElementExtractor* *rangeOut)>

using **TimeRangeExtractorPtr** = *std*::add_pointer_t<void(void *toObject, const void *fromObject, const *TimeElementExtractor* *rangeIn, uint32_t *commitNowOut, uint32_t *exceedsTimeRangeOut)>

Functions

constexpr uint32_t **IdentifierStringToId**(const *std*::string_view id) noexcept

constexpr *std*::array<char, 5> **IdToIdentifierString**(const uint32_t id) noexcept

template<typename **ObjectAPIType**>
inline uint32_t **Packer**(void *toFlatBufferBuilder, const void *fromObject)

template<typename **ObjectAPIType**>
inline void **Unpacker**(void *toObject, const void *fromFlatBuffer)

template<typename **ObjectAPIType**, typename **SubObjectAPIType**>
inline void **TimeElementExtractorDefault**(const void *object, *TimeElementExtractor* *rangeOut) noexcept

template<typename **ObjectAPIType**, typename **SubObjectAPIType**>
inline void **TimeRangeExtractorDefault**(void *toObject, const void *fromObject, const *TimeElementExtractor* *rangeIn, uint32_t *commitNowOut, uint32_t *exceedsTimeRangeAndKeepPacketOut)

template<typename **ObjectAPIType**, typename **SubObjectAPIType**>
constexpr *Type* **makeTypeDefinition**()

template<typename **ObjectAPIType**, typename **SubObjectAPIType**>
constexpr *Type* **makeTypeDefinition**(const *std*::string_view description)

namespace **visualization**

namespace **colors**

Functions

inline cv::Scalar **someNeonColor**(const int32_t someNumber)

Variables

static const cv::Scalar **black** = cv::Scalar(0, 0, 0)

static const cv::Scalar **white** = cv::Scalar(255, 255, 255)

static const cv::Scalar **red** = cv::Scalar(0, 0, 255)

static const cv::Scalar **lime** = cv::Scalar(0, 255, 0)

static const cv::Scalar **blue** = cv::Scalar(255, 0, 0)

static const cv::Scalar **yellow** = cv::Scalar(0, 255, 255)

static const cv::Scalar **cyan** = cv::Scalar(255, 255, 0)

static const cv::Scalar **magenta** = cv::Scalar(255, 0, 255)

static const cv::Scalar **silver** = cv::Scalar(192, 192, 192)

static const cv::Scalar **gray** = cv::Scalar(128, 128, 128)

static const cv::Scalar **navy** = cv::Scalar(128, 0, 0)

static const cv::Scalar **green** = cv::Scalar(0, 128, 0)

static const cv::Scalar **iniBlue** = cv::Scalar(183, 93, 0)

static const cv::Scalar **darkGrey** = cv::Scalar(43, 43, 43)

static const auto **iniblue** = *iniBlue*

static const auto **darkgrey** = *darkGrey*

static const *std*::vector<cv::Scalar> **neonPalette** = {cv::Scalar(255, 111, 0), cv::Scalar(239, 244, 19), cv::Scalar(0, 255, 104), cv::Scalar(0, 255, 250), cv::Scalar(0, 191, 255), cv::Scalar(0, 191, 255), cv::Scalar(92, 0, 255)}

namespace **flatbuffers**

namespace **fmt**

fmt formatting support, adds automatic direct formatting support for common data structures:

- std::filesystem::path
- std::vector<T>

namespace **std**

file **calibration_set.hpp**

#include "../core/utils.hpp"#include "../exception/exception.hpp"#include "calibrations/camera_calibration.hpp"#include "calibrations/imu_calibration.hpp"#include "calibrations/stereo_calibration.hpp"#include <boost/algorithm/string.hpp>#include <boost/property_tree/json_parser.hpp>#include <boost/property_tree/ptree.hpp>#include <opencv2/core.hpp>#include <iostream>#include <map>#include <regex>#include <vector>

file **camera_calibration.hpp**

*#include "../../external/fmt_compat.hpp"#include "../../exception/exception.hpp"#include "../camera_geometry.hpp"#include <Eigen/Core>#include <boost/property_tree/ptree.hpp>#include <opencv2/core.hpp>#include <optional>#include *

file **imu_calibration.hpp**

*#include "camera_calibration.hpp"#include *

file **stereo_calibration.hpp**

#include "camera_calibration.hpp"

file **camera_geometry.hpp**

#include "../core/core.hpp"#include "../data/timed_keypoint_base.hpp"#include "../exception/exception.hpp"#include "../kinematics/linear_transformer.hpp"#include <Eigen/Core>#include <opencv2/calib3d.hpp>#include <opencv2/core.hpp>#include <opencv2/core/eigen.hpp>#include <cmath>#include <vector>

file **stereo_geometry.hpp**

#include "../core/utils.hpp"#include "../data/depth_frame_base.hpp"#include "calibrations/camera_calibration.hpp"#include "camera_geometry.hpp"#include <opencv2/imgproc.hpp>

file **mean_shift.hpp**

#include "mean_shift/eigen_matrix_adaptor.hpp"#include "mean_shift/event_store_adaptor.hpp"

file **eigen_matrix_adaptor.hpp**

#include "../../containers/kd_tree.hpp"#include "kernel.hpp"#include <Eigen/Dense>#include <optional>#include <random>#include <vector>

file **eigen_matrix_adaptor.hpp**

#include "../../external/nanoflann/nanoflann.hpp"#include <Eigen/Dense>#include <memory>

file **event_store_adaptor.hpp**

#include "../../containers/kd_tree.hpp"#include "kernel.hpp"#include <optional>#include <random>#include <vector>

file **event_store_adaptor.hpp**

#include "../../external/nanoflann/nanoflann.hpp"#include "../../core/core.hpp"#include "../../data/timed_keypoint_base.hpp"#include <opencv2/core.hpp>#include <memory>

file **kernel.hpp**

#include <cmath>#include <concepts>

file **kd_tree.hpp**

#include "kd_tree/eigen_matrix_adaptor.hpp"#include "kd_tree/event_store_adaptor.hpp"

file **concepts.hpp**

#include "../data/bounding_box_base.hpp"#include "../data/depth_event_base.hpp"#include "../data/depth_frame_base.hpp"#include "../data/event_base.hpp"#include "../data/frame_base.hpp"#include "../data/imu_base.hpp"#include "../data/landmark_base.hpp"#include "../data/pose_base.hpp"#include "../data/timed_keypoint_base.hpp"#include "../data/trigger_base.hpp"#include <Eigen/Core>#include <boost/callable_traits.hpp>#include <opencv2/core.hpp>#include <concepts>#include <iterator>

file **core.hpp**

#include "../data/depth_event_base.hpp"#include "../data/event_base.hpp"#include "../data/frame_base.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "concepts.hpp"#include "stream_slicer.hpp"#include "time.hpp"#include <Eigen/Dense>#include <opencv2/core.hpp>#include <opencv2/core/eigen.hpp>#include <algorithm>#include <functional>#include <iostream>#include <map>#include <memory>#include <numeric>#include <optional>#include <vector>

file **dvassert.hpp**

#include "../external/fmt_compat.hpp"#include "../external/source_location_compat.hpp"#include <boost/stacktrace.hpp>#include <cstdlib>#include <filesystem>#include <string_view>

file **event.hpp**

#include "core.hpp"#include "filters.hpp"

file **event_color.hpp**

#include "../data/event_base.hpp"

file **filters.hpp**

#include "../core/frame.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include <valarray>

file **frame.hpp**

#include "frame/accumulator.hpp"#include "frame/accumulator_base.hpp"#include "frame/edge_map_accumulator.hpp"

file **accumulator.hpp**

#include "accumulator_base.hpp"

file **accumulator_base.hpp**

#include "../core.hpp"

file **edge_map_accumulator.hpp**

#include "accumulator_base.hpp"

file **multi_stream_slicer.hpp**

#include "../data/frame_base.hpp"#include "../data/imu_base.hpp"#include "../data/trigger_base.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "core.hpp"#include "stream_slicer.hpp"#include <unordered_map>#include <variant>

file **stereo_event_stream_slicer.hpp**

#include "core.hpp"

file **stream_slicer.hpp**

#include "../exception/exceptions/generic_exceptions.hpp"#include "concepts.hpp"#include "time_window.hpp"#include "utils.hpp"#include <functional>#include <map>

file **time.hpp**

#include <chrono>

file **time_window.hpp**

#include "time.hpp"

file **utils.hpp**

#include "../external/compare_compat.hpp"#include "../external/fmt_compat.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "concepts.hpp"#include "dvassert.hpp"#include "time.hpp"#include "time_window.hpp"#include <algorithm>#include <array>#include <cerrno>#include <cinttypes>#include <cstddef>#include <cstdint>#include <cstdlib>#include <cstring>#include <filesystem>#include <functional>#include <memory>#include <stdexcept>#include <string>#include <string_view>#include <type_traits>#include <utility>#include <vector>

file **utils.hpp**

#include <opencv2/calib3d.hpp>

file **utils.hpp**

#include "../../core/utils.hpp"#include "../../data/bounding_box_base.hpp"#include "../../data/depth_event_base.hpp"#include "../../data/depth_frame_base.hpp"#include "../../data/event_base.hpp"#include "../../data/frame_base.hpp"#include "../../data/imu_base.hpp"#include "../../data/landmark_base.hpp"#include "../../data/pose_base.hpp"#include "../../data/timed_keypoint_base.hpp"#include "../../data/trigger_base.hpp"#include "../../data/types.hpp"#include "../data/FileDataTable.hpp"#include "../data/IOHeader.hpp"#include "io_data_buffer.hpp"#include "io_statistics.hpp"#include <string_view>

file **boost_geometry_interop.hpp**

#include "bounding_box_base.hpp"#include "event_base.hpp"#include "timed_keypoint_base.hpp"#include <boost/geometry/core/cs.hpp>#include <boost/geometry/geometries/register/box.hpp>#include <boost/geometry/geometries/register/point.hpp>#include <boost/geometry/geometry.hpp>#include <opencv2/core.hpp>

file **bounding_box_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cstring.hpp*"#include "*cvector.hpp*"*

Variables

VT_TIMESTAMP = 4

VT_TOPLEFTX = 6

VT_TOPLEFTY = 8

VT_BOTTOMRIGHTX = 10

VT_BOTTOMRIGHTY = 12

VT_CONFIDENCE = 14

file **cptriterator.hpp**

#include <cinttypes>#include <cstddef>#include <cstdint>#include <cstdlib>#include <iterator>#include <type_traits>

file **cstring.hpp**

*#include "../external/compare_compat.hpp"#include "../external/fmt_compat.hpp"#include "../core/dvassert.hpp"#include "*cptriterator.hpp*"#include <array>#include <concepts>#include <filesystem>#include <limits>#include <stdexcept>#include <string>#include <string_view>*

file **cvector.hpp**

*#include "../external/compare_compat.hpp"#include "../external/fmt_compat.hpp"#include "../core/dvassert.hpp"#include "*cptriterator.hpp*"#include <array>#include <concepts>#include <limits>#include #include <stdexcept>#include <string_view>*

file **depth_event_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cvector.hpp*"*

file **depth_frame_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cvector.hpp*"*

Variables

VT_TIMESTAMP = 4

VT_SIZEX = 6

VT_SIZEY = 8

VT_MINDEPTH = 10

VT_MAXDEPTH = 12

VT_STEP = 14

file **event_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cvector.hpp*"*

file **frame_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "../core/time.hpp"#include "*cvector.hpp*"#include <opencv2/core/mat.hpp>#include <ostream>*

Variables

VT_TIMESTAMP = 4

VT_TIMESTAMPSTARTOFFRAME = 6

VT_TIMESTAMPENDOFFRAME = 8

VT_TIMESTAMPSTARTOFEXPOSURE = 10

VT_TIMESTAMPENDOFEXPOSURE = 12

VT_FORMAT = 14

VT_SIZEX = 16

VT_SIZEY = 18

VT_POSITIONX = 20

VT_POSITIONY = 22

VT_PIXELS = 24

VT_EXPOSURE = 26

file **generate.hpp**

#include "../core/core.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "../visualization/colors.hpp"#include <opencv2/imgproc.hpp>#include <random>

file **geometry_types_base.hpp**

#include "../external/flatbuffers/flatbuffers.h"

file **imu_base.hpp**

#include "../external/flatbuffers/flatbuffers.h"#include "cvector.hpp"#include <Eigen/Core>#include <numbers>#include <ostream>

Variables

VT_TIMESTAMP = 4

VT_TEMPERATURE = 6

VT_ACCELEROMETERX = 8

VT_ACCELEROMETERY = 10

VT_ACCELEROMETERZ = 12

VT_GYROSCOPEX = 14

VT_GYROSCOPEY = 16

VT_GYROSCOPEZ = 18

VT_MAGNETOMETERX = 20

VT_MAGNETOMETERY = 22

file **landmark_base.hpp**

#include "../external/flatbuffers/flatbuffers.h"#include "cstring.hpp"#include "cvector.hpp"#include "geometry_types_base.hpp"

Variables

VT_TRACKID = 4

VT_CAMERAID = 6

VT_CAMERANAME = 8

VT_PT = 4

VT_ID = 6

VT_TIMESTAMP = 8

VT_DESCRIPTOR = 10

VT_DESCRIPTORTYPE = 12

VT_COVARIANCE = 14

VT_ELEMENTS = 4

file **pose_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cstring.hpp*"#include "*geometry_types_base.hpp*"*

Variables

VT_TIMESTAMP = 4

VT_TRANSLATION = 6

VT_ROTATION = 8

VT_REFERENCEFRAME = 10

file **timed_keypoint_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cvector.hpp*"#include "*geometry_types_base.hpp*"*

Variables

VT_PT = 4

VT_SIZE = 6

VT_ANGLE = 8

VT_RESPONSE = 10

VT_OCTAVE = 12

VT_CLASS_ID = 14

file **trigger_base.hpp**

*#include "../external/flatbuffers/flatbuffers.h"#include "*cvector.hpp*"*

Variables

VT_TIMESTAMP = 4

file **types.hpp**

#include "../external/flatbuffers/flatbuffers.h"#include "../core/utils.hpp*"#include "*../exception/exceptions/generic_exceptions.hpp*"*

file **utilities.hpp**

#include "../core/core.hpp*"#include "*../kinematics/transformation.hpp*"#include "*depth_event_base.hpp*"#include "*depth_frame_base.hpp*"#include "*event_base.hpp*"#include "*pose_base.hpp*"#include "*timed_keypoint_base.hpp*"#include <opencv2/core.hpp>*

file **semi_dense_stereo_matcher.hpp**

#include "../camera/stereo_geometry.hpp*"#include "*../core/concepts.hpp*"#include "*../core/frame.hpp*"#include "*utils.hpp*"*

file **sparse_event_block_matcher.hpp**

#include "../camera/stereo_geometry.hpp*"#include "*../core/filters.hpp*"#include "*../core/frame.hpp*"#include <opencv2/imgproc.hpp>*

file **exception.hpp**

*#include "*exceptions/directory_exceptions.hpp*"#include "*exceptions/file_exceptions.hpp*"#include "*exceptions/generic_exceptions.hpp*"#include "*exceptions/io_exceptions.hpp*"#include "*exceptions/type_exceptions.hpp*"*

file **exception_base.hpp**

*#include "../external/source_location_compat.hpp"#include "*internal.hpp*"#include <boost/core/demangle.hpp>#include <boost/stacktrace.hpp>#include <filesystem>#include <string>*

file **directory_exceptions.hpp**

#include "../exception_base.hpp"

file **file_exceptions.hpp**

#include "../exception_base.hpp"

file **generic_exceptions.hpp**

#include "../exception_base.hpp"

file **io_exceptions.hpp**

#include "../../data/cstring.hpp"#include "../exception_base.hpp"

file **type_exceptions.hpp**

#include "../../data/cstring.hpp"#include "../exception_base.hpp"

file **internal.hpp**

#include "../external/fmt_compat.hpp"#include <concepts>#include <string>

file **arc_corner_detector.hpp**

#include "../core/concepts.hpp"#include "../core/core.hpp"#include "../data/timed_keypoint_base.hpp"#include <Eigen/Dense>#include <opencv2/core.hpp>

file **event_blob_detector.hpp**

#include "../core/event.hpp"#include "../core/frame/accumulator.hpp"#include "../data/utilities.hpp"#include <opencv2/opencv.hpp>#include <atomic>#include <utility>

file **event_combined_lk_tracker.hpp**

#include "../core/core.hpp"#include "../core/frame.hpp"#include "../data/utilities.hpp"#include "image_feature_lk_tracker.hpp"

file **event_feature_lk_tracker.hpp**

#include "../core/frame.hpp"#include "image_feature_lk_tracker.hpp"

file **feature_detector.hpp**

#include "../core/concepts.hpp"#include "../core/core.hpp"#include "../data/timed_keypoint_base.hpp"#include "../data/utilities.hpp"#include "event_blob_detector.hpp"#include "image_pyramid.hpp"#include "keypoint_resampler.hpp"#include <opencv2/core.hpp>#include <opencv2/features2d.hpp>

file **feature_tracks.hpp**

#include "../core/utils.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "../visualization/colors.hpp"#include "tracker_base.hpp"

file **image_feature_lk_tracker.hpp**

#include "../data/utilities.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "../kinematics/motion_compensator.hpp"#include "image_pyramid.hpp"#include "redetection_strategy.hpp"#include "tracker_base.hpp"#include <utility>

file **image_pyramid.hpp**

#include "../data/frame_base.hpp"#include <opencv2/core.hpp>#include <opencv2/video.hpp>#include <memory>

file **keypoint_resampler.hpp**

#include "../core/concepts.hpp"#include "../data/boost_geometry_interop.hpp"#include <boost/geometry/geometry.hpp>#include <boost/geometry/index/rtree.hpp>

file **mean_shift_tracker.hpp**

#include "../core/core.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "feature_detector.hpp"#include "redetection_strategy.hpp"#include "tracker_base.hpp"

file **redetection_strategy.hpp**

#include "tracker_base.hpp"

file **tracker_base.hpp**

#include "feature_detector.hpp"

file **imgproc.hpp**

#include "../external/fmt_compat.hpp"#include <Eigen/Dense>#include <opencv2/core.hpp>#include <opencv2/opencv.hpp>#include <optional>

file **rotation-integrator.hpp**

#include "../core/concepts.hpp"#include "../data/imu_base.hpp"#include "../kinematics/transformation.hpp"#include <Eigen/Geometry>#include <numbers>

file **camera_capture.hpp**

#include "../core/core.hpp"#include "../core/utils.hpp"#include "../data/event_base.hpp"#include "../data/frame_base.hpp"#include "../data/imu_base.hpp"#include "../data/trigger_base.hpp"#include "../exception/exception.hpp"#include "camera_input_base.hpp"#include "data_read_handler.hpp"#include "discovery.hpp"#include <boost/lockfree/spsc_queue.hpp>#include <opencv2/imgproc.hpp>#include <atomic>#include <functional>#include <future>#include <thread>

file **camera_input_base.hpp**

#include "../core/core.hpp"#include "../data/cvector.hpp"#include "../data/event_base.hpp"#include "../data/frame_base.hpp"#include "../data/imu_base.hpp"#include "../data/trigger_base.hpp"#include <opencv2/core.hpp>#include <optional>#include <string>

file **camera_output_base.hpp**

#include "../core/core.hpp"

file **compression_support.hpp**

#include "../../external/fmt_compat.hpp"#include "../../core/utils.hpp"#include "../data/IOHeader.hpp"#include "../support/io_data_buffer.hpp"#include <lz4.h>#include <lz4frame.h>#include <lz4hc.h>#include <memory>#include <vector>#include <zstd.h>

Defines

LZ4F_HEADER_SIZE_MAX

ZSTD_CLEVEL_DEFAULT

file **decompression_support.hpp**

#include "../../external/fmt_compat.hpp"#include "../../core/utils.hpp"#include "../data/IOHeader.hpp"#include <lz4.h>#include <lz4frame.h>#include <lz4hc.h>#include <memory>#include <vector>#include <zstd.h>

Defines

LZ4F_HEADER_SIZE_MAX

ZSTD_CLEVEL_DEFAULT

file **FileDataTable.hpp**

#include "../../external/flatbuffers/flatbuffers.h"#include "../../data/cvector.hpp"

Variables

VT_BYTEOFFSET = 4

VT_PACKETINFO = 6

VT_NUMELEMENTS = 8

VT_TIMESTAMPSTART = 10

file **IOHeader.hpp**

#include "../../external/flatbuffers/flatbuffers.h"#include "../../data/cstring.hpp"

Variables

VT_COMPRESSION = 4

VT_DATATABLEPOSITION = 6

file **data_read_handler.hpp**

#include "../core/core.hpp"#include "../core/frame.hpp"#include "../data/imu_base.hpp"#include "../data/trigger_base.hpp"#include <functional>#include <optional>#include <variant>

file **discovery.hpp**

#include "../exception/exceptions/generic_exceptions.hpp"#include <libcaercpp/devices/device_discover.hpp>

file **mono_camera_recording.hpp**

#include "../core/frame.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "camera_input_base.hpp"#include "data_read_handler.hpp"#include "read_only_file.hpp"#include <functional>#include <optional>

file **mono_camera_writer.hpp**

#include "../core/core.hpp"#include "../core/frame.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "camera_capture.hpp"#include "reader.hpp"#include "support/utils.hpp"#include "support/xml_config_io.hpp"#include "write_only_file.hpp"

file **encrypt.hpp**

#include <boost/asio/ssl.hpp>#include <filesystem>#include <optional>

file **socket_base.hpp**

#include <boost/asio.hpp>

file **tcp_tls_socket.hpp**

#include "encrypt.hpp"#include "socket_base.hpp"#include <deque>#include <mutex>#include <utility>

file **unix_socket.hpp**

#include "socket_base.hpp"#include <deque>#include <mutex>#include <utility>

file **write_ordered_socket.hpp**

#include "socket_base.hpp"#include <deque>#include <functional>#include <utility>

file **network_reader.hpp**

#include "camera_input_base.hpp"#include "network/encrypt.hpp"#include "network/tcp_tls_socket.hpp"#include "network/unix_socket.hpp"#include "reader.hpp"#include <boost/lockfree/spsc_queue.hpp>

file **network_writer.hpp**

#include "camera_output_base.hpp"#include "network/socket_base.hpp"#include "network/tcp_tls_socket.hpp"#include "network/unix_socket.hpp"#include "network/write_ordered_socket.hpp"#include "stream.hpp"#include "support/utils.hpp"#include "writer.hpp"#include <boost/lockfree/spsc_queue.hpp>#include <utility>

file **read_only_file.hpp**

#include "reader.hpp"#include "simplefile.hpp"

file **reader.hpp**

#include "compression/decompression_support.hpp"#include "stream.hpp"#include <boost/endian.hpp>#include <optional>#include <unordered_map>#include <utility>

file **simplefile.hpp**

#include "../core/utils.hpp"#include "../data/cstring.hpp"#include "../data/cvector.hpp"#include "../exception/exceptions/file_exceptions.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include <boost/nowide/cstdio.hpp>#include <cstdio>#include <filesystem>#include <limits>

file **stereo_camera_recording.hpp**

#include "mono_camera_recording.hpp"

file **stereo_camera_writer.hpp**

#include "mono_camera_writer.hpp"#include "stereo_capture.hpp"

file **stereo_capture.hpp**

#include "camera_capture.hpp"

file **stream.hpp**

#include "support/utils.hpp"#include "support/xml_config_io.hpp"#include <opencv2/core.hpp>#include <optional>

file **io_data_buffer.hpp**

#include "../data/FileDataTable.hpp"#include <vector>

file **io_statistics.hpp**

#include <cstdint>

file **xml_config_io.hpp**

#include "../../core/utils.hpp"#include <boost/property_tree/ptree.hpp>#include <boost/property_tree/xml_parser.hpp>#include <map>#include <sstream>#include <variant>

file **write_only_file.hpp**

#include "simplefile.hpp"#include "writer.hpp"#include <atomic>#include <mutex>#include <queue>#include <thread>

file **writer.hpp**

#include "compression/compression_support.hpp"#include "support/utils.hpp"#include <iostream>#include <memory>

file **linear_transformer.hpp**

#include "transformation.hpp"#include <Eigen/Dense>#include <Eigen/StdVector>#include <boost/circular_buffer.hpp>#include <optional>

file **motion_compensator.hpp**

#include "../core/concepts.hpp"#include "../core/frame.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "../measurements/depth.hpp"#include "linear_transformer.hpp"#include "pixel_motion_predictor.hpp"

file **pixel_motion_predictor.hpp**

#include "../camera/camera_geometry.hpp"#include <utility>

file **transformation.hpp**

#include "../core/concepts.hpp"#include <Eigen/Core>#include <opencv2/core/eigen.hpp>

file **depth.hpp**

#include <cstdint>

file **background_activity_noise_filter.hpp**

#include "../core/filters.hpp"

file **fast_decay_noise_filter.hpp**

#include "../core/filters.hpp"

file **contrast_maximization_rotation.hpp**

#include "../camera/camera_geometry.hpp"#include "../core/core.hpp"#include "../imu/rotation-integrator.hpp"#include "../kinematics/motion_compensator.hpp"#include "../optimization/optimization_functor.hpp"

file **contrast_maximization_translation_and_depth.hpp**

#include "../camera/camera_geometry.hpp"#include "../core/core.hpp"#include "../imu/rotation-integrator.hpp"#include "../kinematics/motion_compensator.hpp"#include "../optimization/optimization_functor.hpp"

file **contrast_maximization_wrapper.hpp**

#include "../core/concepts.hpp"#include "../optimization/optimization_functor.hpp"#include <memory>#include <unsupported/Eigen/NonLinearOptimization>#include <unsupported/Eigen/NumericalDiff>

file **optimization_functor.hpp**

#include <Eigen/Dense>

file **processing.hpp**

#include "camera/calibration_set.hpp"#include "camera/calibrations/camera_calibration.hpp"#include "camera/calibrations/imu_calibration.hpp"#include "camera/calibrations/stereo_calibration.hpp"#include "camera/camera_geometry.hpp"#include "camera/stereo_geometry.hpp"#include "cluster/mean_shift.hpp"#include "containers/kd_tree.hpp"#include "core/core.hpp"#include "core/event.hpp"#include "core/event_color.hpp"#include "core/filters.hpp"#include "core/frame.hpp"#include "core/multi_stream_slicer.hpp"#include "core/stereo_event_stream_slicer.hpp"#include "core/stream_slicer.hpp"#include "core/time.hpp"#include "core/utils.hpp"#include "data/boost_geometry_interop.hpp"#include "data/bounding_box_base.hpp"#include "data/cstring.hpp"#include "data/cvector.hpp"#include "data/depth_event_base.hpp"#include "data/depth_frame_base.hpp"#include "data/event_base.hpp"#include "data/frame_base.hpp"#include "data/generate.hpp"#include "data/geometry_types_base.hpp"#include "data/imu_base.hpp"#include "data/landmark_base.hpp"#include "data/pose_base.hpp"#include "data/timed_keypoint_base.hpp"#include "data/trigger_base.hpp"#include "data/types.hpp"#include "data/utilities.hpp"#include "depth/semi_dense_stereo_matcher.hpp"#include "depth/sparse_event_block_matcher.hpp"#include "exception/exception.hpp"#include "features/arc_corner_detector.hpp"#include "features/event_blob_detector.hpp"#include "features/event_combined_lk_tracker.hpp"#include "features/event_feature_lk_tracker.hpp"#include "features/feature_detector.hpp"#include "features/feature_tracks.hpp"#include "features/image_feature_lk_tracker.hpp"#include "features/image_pyramid.hpp"#include "features/keypoint_resampler.hpp"#include "features/mean_shift_tracker.hpp"#include

"features/redetection_strategy.hpp"*#include* "features/tracker_base.hpp"*#include* "imgproc/imgproc.hpp"*#include* "imu/rotation-integrator.hpp"*#include* "io/camera_capture.hpp"*#include* "io/camera_output_base.hpp"*#include* "io/data_read_handler.hpp"*#include* "io/discovery.hpp"*#include* "io/mono_camera_recording.hpp"*#include* "io/mono_camera_writer.hpp"*#include* "io/network/tcp_tls_socket.hpp"*#include* "io/network_reader.hpp"*#include* "io/network_writer.hpp"*#include* "io/read_only_file.hpp"*#include* "io/reader.hpp"*#include* "io/simplefile.hpp"*#include* "io/stereo_camera_recording.hpp"*#include* "io/stereo_camera_writer.hpp"*#include* "io/stereo_capture.hpp"*#include* "io/write_only_file.hpp"*#include* "io/writer.hpp"*#include* "kinematics/linear_transformer.hpp"*#include* "kinematics/motion_compensator.hpp"*#include* "kinematics/pixel_motion_predictor.hpp"*#include* "kinematics/transformation.hpp"*#include* "measurements/depth.hpp"*#include* "noise/background_activity_noise_filter.hpp"*#include* "noise/fast_decay_noise_filter.hpp"*#include* "optimization/contrast_maximization_rotation.hpp"*#include* "optimization/contrast_maximization_translation_and_depth.hpp"*#include* "optimization/contrast_maximization_wrapper.hpp"*#include* "version.hpp"*#include* "visualization/colors.hpp"*#include* "visualization/event_visualizer.hpp"*#include* "visualization/pose_visualizer.hpp"

file **version.hpp**

#include <string_view>

Defines

DV_PROCESSING_VERSION_MAJOR

dv-processing version (MAJOR * 10000 + MINOR * 100 + PATCH).

DV_PROCESSING_VERSION_MINOR

DV_PROCESSING_VERSION_PATCH

DV_PROCESSING_VERSION

DV_PROCESSING_NAME_STRING

dv-processing name string.

DV_PROCESSING_VERSION_STRING

dv-processing version string.

file **colors.hpp**

#include <opencv2/core.hpp>

file **event_visualizer.hpp**

#include "../core/core.hpp"*#include* "../core/utils.hpp"*#include* "../exception/exceptions/generic_exceptions.hpp"*#include* "colors.hpp"

file **events_visualizer.hpp**

#include "event_visualizer.hpp"

file **pose_visualizer.hpp**

#include "../core/concepts.hpp"#include "../core/utils.hpp"#include "../data/frame_base.hpp"#include "../data/landmark_base.hpp"#include "../exception/exceptions/generic_exceptions.hpp"#include "../kinematics/linear_transformer.hpp"#include "../visualization/colors.hpp"#include <Eigen/Core>#include <Eigen/Geometry>#include <fmt/format.h>#include <opencv2/opencv.hpp>#include <chrono>#include <numbers>

page **deprecated**

Member *dv::Accumulator::isRectifyPolarity* () const

Use *isIgnorePolarity()* method instead.

Member *dv::Accumulator::setRectifyPolarity* (bool rectifyPolarity)

Use *setIgnorePolarity()* method instead.

Member *dv::EdgeMapAccumulator::getContribution* () const

Use *getEventContribution()* method instead.

Member *dv::EdgeMapAccumulator::getNeutralValue* () const

Use *getNeutralPotential()* method instead.

Member *dv::EdgeMapAccumulator::setContribution* (const float contribution_)

Use *setEventContribution()* method instead.

Member *dv::EdgeMapAccumulator::setNeutralValue* (const float neutralValue_)

Use *setNeutralPotential()* method instead.

Member *dv::features::ImageFeatureLKTracker::setRedectionStrategy* (RedetectionStrategy::UniquePtr redetectionStrategy)

Use setRedetectionStrategy instead

Member *dv::features::RedetectionStrategy::decideRedection* (const *dv::features::TrackerBase* &tracker)

Use decideRedetection instead

Member *dv::io::CameraCapture::isConnected* () const

Please use `isRunning()` method instead.

Member *dv::StreamSlicer< PacketType >::doEveryNumberOfEvents* (const size_t n, std::function< void(PacketType &)> callback)

Use *doEveryNumberOfElements()* method instead.

Member *dv::StreamSlicer< PacketType >::doEveryTimeInterval* (const int64_t microseconds, std::function< void(const PacketType &)> callback)

Please pass interval parameter using dv::Duration.

Member *dv::StreamSlicer< PacketType >::modifyTimeInterval* (const int jobId, const int64_t timeInterval)

Please pass time interval as dv::Duration instead.

Member *dv::TimeSurfaceBase< EventStoreType, ScalarType >::empty* () const noexcept

Use *isEmpty()* instead.

dir **/builds/inivation/dv/dv-processing/include/dv-processing/camera/calibrations**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/camera**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/cluster**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/io/compression**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/containers**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/core**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/data**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/io/data**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/depth**

dir **/builds/inivation/dv/dv-processing/include/dv-processing**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/exception**

dir
/builds/inivation/dv/dv-processing/include/dv-processing/exception/exceptions

dir **/builds/inivation/dv/dv-processing/include/dv-processing/features**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/core/frame**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/imgproc**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/imu**

dir **/builds/inivation/dv/dv-processing/include**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/io**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/containers/kd_tree**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/kinematics**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/cluster/mean_shift**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/measurements**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/io/network**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/noise**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/optimization**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/io/support**

dir **/builds/inivation/dv/dv-processing/include/dv-processing/visualization**

CHAPTER

SEVEN

HELP

In case of technical issues or any problems, please visit the support page[29]. Any issues in the documentation or the code can be reported to our gitlab issue tracker[30].

[29] https://docs.inivation.com/help/support.html
[30] https://gitlab.com/inivation/dv/dv-processing/-/issues

INDEX

D

F

L

S

Z