VisionLabs
MACHINES CAN SEE

VisionLabs FaceEngine Handbook

written for LUNA SDK version 5.32.0

Contents

Introduction **11**

Editions and Platforms **12**

1 Core Concepts **13**

1.1 SDK workflow . . . 13
1.1.1 Object lifetime . . . 13
1.1.2 Threading . . . 14
1.1.3 Detailed constraints . . . 15
1.2 Common Interfaces and Types . . . 16
1.2.1 Reference Counted Interface . . . 16
1.2.2 Automatic reference counting . . . 17
1.2.2.1 Referencing - without acquiring ownership of object lifetime . . . 17
1.2.2.2 Acquiring - own object lifetime . . . 18
1.2.3 Serializable object interface . . . 19
1.2.4 Auxiliary types . . . 19
1.2.4.1 Image type . . . 19
1.3 Beta Mode . . . 20

2 FaceEngine Structure Overview **21**

3 Core Facility **22**

3.1 Common Interfaces . . . 22
3.1.1 Face Engine Object . . . 22
3.1.2 Settings Provider . . . 22
3.2 Helper Interfaces . . . 22
3.2.1 Archive Interface . . . 22
3.3 Sensor type . . . 23
3.4 Data Paths . . . 23
3.4.1 Model Data . . . 23
3.4.2 Configuration Data . . . 23

4 Detection facility **25**

4.1 Overview . . . 25
4.2 Detection structure . . . 25
4.3 Face Detection . . . 25
4.3.1 Image coordinate system . . . 25
4.3.2 Face detection . . . 26
4.3.3 Redetect method . . . 26

4.3.4 Orientation Estimation . . . 26
4.3.5 Detector variants . . . 27
4.3.6 FaceDetV2 Configuration . . . 29
4.3.7 FaceDetV3 Configuration . . . 29
4.3.8 Face Alignment . . . 29
4.3.8.1 Five landmarks . . . 29
4.3.8.2 Sixty-eight landmarks . . . 30
4.4 Face Landmarks Detector . . . 32
4.5 Human Detection . . . 33
4.5.1 Image coordinate system . . . 33
4.5.2 Human body detection . . . 33
4.5.3 Constraints . . . 33
4.5.4 Camera position requirements . . . 34
4.5.5 Human body redetection . . . 35
4.5.6 Human keypoints . . . 36
4.5.7 Main results of each detection . . . 37
4.5.8 HumanFace redetection . . . 38
4.5.9 Performance . . . 38
4.5.10 Main results . . . 38
4.5.11 minFaceSize . . . 39
4.6 Head Detection . . . 40
4.6.1 Image coordinate system . . . 40
4.6.2 Main results . . . 40
4.6.3 minHeadSize . . . 41

5 Image Warping **42**

6 Parameter Estimation Facility **43**
6.1 Overview . . . 43
6.2 Use cases . . . 43
6.2.1 ISO estimation . . . 43
6.3 Best shot selection functionality . . . 44
6.3.1 BestShotQuality Estimation . . . 44
6.3.2 Image Quality Estimation . . . 48
6.3.3 Image Subjective Quality Estimation . . . 55
6.4 Attributes estimation functionality . . . 61
6.4.1 Face Attribute Estimation . . . 61
6.4.2 Credibility Check Estimation . . . 64
6.5 Facial Hair Estimation . . . 67
6.6 Natural Light Estimation . . . 70

6.7 Fish Eye Estimation . . . 73
6.8 Eyebrows Estimation . . . 76
6.9 Portrait Style Estimation . . . 78
6.10 DynamicRange Estimation . . . 81
6.11 Headwear Estimation . . . 83
6.12 Background Estimation . . . 87
6.13 Grayscale, color or infrared Estimation . . . 90
6.14 Face features extraction functionality . . . 92
6.14.1 Eyes Estimation . . . 92
6.14.2 Red Eyes Estimation . . . 95
6.14.3 Gaze Estimation . . . 98
6.15 Head Pose Estimation . . . 100
6.16 Approximate Garbage Score Estimation (AGS) . . . 102
6.16.1 Glasses Estimation . . . 103
6.16.2 Overlap Estimation . . . 105
6.17 Emotion estimation functionality . . . 106
6.17.1 Emotions Estimation . . . 106
6.18 Mouth Estimation Functionality . . . 108
6.19 Face Occlusion Estimation Functionality . . . 111
6.20 DeepFake estimation functionality . . . 114
6.20.1 LivenessFlyingFaces Estimation . . . 117
6.20.2 LivenessRGBM Estimation . . . 119
6.20.3 Depth Liveness Estimation (LivenessDepthEstimator) . . . 121
6.20.4 Depth and RGB OneShotLiveness estimation . . . 123
6.20.5 Depth liveness estimation (DepthLivenessEstimator) . . . 126
6.20.6 LivenessOneShotRGB Estimation . . . 129
6.20.6.1 Usage example . . . 132
6.20.7 NIR Liveness estimation . . . 134
6.21 Personal Protection Equipment Estimation . . . 136
6.22 Medical Mask Estimation Functionality . . . 139
6.22.1 MedicalMaskEstimator thresholds . . . 140
6.22.2 MedicalMask enumeration . . . 140
6.22.3 MedicalMaskEstimation structure . . . 141
6.22.4 MedicalMaskExtended enumeration . . . 142
6.22.5 MedicalMaskEstimationExtended structure . . . 142
6.22.6 Filtration parameters . . . 143
6.23 Human Attribute Estimation . . . 145
6.24 Crowd Estimation . . . 157
6.25 Fights Estimation . . . 159

6.26 ImageModification Estimation . . . 163

7 Descriptor Processing Facility **165**

7.1 Overview . . . 165

7.1.1 Person Identification Task . . . 165

7.1.2 Person Reidentification Task . . . 165

7.2 Descriptor . . . 167

7.2.1 Descriptor Versions . . . 167

7.2.1.1 Face descriptor . . . 167

7.2.1.2 Human descriptor . . . 168

7.2.2 Descriptor Batch . . . 168

7.2.3 Descriptor Extraction . . . 169

7.2.4 Descriptor Matching . . . 170

7.2.5 Descriptor Indexing . . . 171

7.2.5.1 Using HNSW . . . 171

7.2.5.2 Index serialization . . . 172

7.2.5.3 Dynamic index evaluation scheme. This feature is experimental. Backward compatibility is not guaranteed. . . . 173

8 System Requirements **176**

8.1 Windows OS installations . . . 176

8.2 Linux OS installations . . . 176

9 Hardware requirements **177**

9.1 Server / PC installations . . . 177

9.1.1 General considerations . . . 177

9.1.2 CPU requirements . . . 179

9.1.3 GPU requirements . . . 180

9.1.4 The number of SDK threads while using GPU . . . 180

9.1.5 NPU requirements . . . 180

9.1.6 RAM requirements . . . 181

9.1.7 Storage requirements . . . 181

9.1.8 Approaches to software design targeting different hardware . . . 181

9.1.8.1 CPU . . . 181

9.1.8.2 GPU/NPU . . . 182

9.1.9 Requirements for GPU acceleration . . . 183

9.2 Embedded installations . . . 184

9.2.1 CPU requirements . . . 184

10 Migration guide **185**

10.1 Overview . . . 185

10.2 v.5.32.0 185
10.2.1 IQualityEstimator 185
10.2.2 ILicense::loadFromFile(const char* path) 186
10.3 v.5.31.0 186
10.3.1 IDeepFakeEstimator 186
10.4 v.5.27.0 187
10.4.1 Multithreading usage of factory functions (for example: createAGSEstimator, createHeadWearEstimator…) 187
10.5 v.5.24.0 187
10.5.1 IDetector 187
10.6 v.5.23.0 187
10.6.1 IImageTransfer 187
10.6.2 IDetector 187
10.7 v.5.22.0 188
10.7.1 IHeadPoseEstimator 188
10.7.2 IHeadPoseEstimator and IAGSEstimator 188
10.8 v.5.20.0 188
10.8.1 ILivenessFlowEstimator 188
10.9 v.5.19.0 188
10.9.1 ILivenessFlowEstimator 188
10.10 v.5.18.0 188
10.10.1 IChildEstimator 188
10.10.2 IHeadAndShouldersLivenessEstimator 188
10.11 v.5.17.0 189
10.11.1 IHeadAndShouldersLivenessEstimator 189
10.11.2 IChildEstimator 189
10.11.3 Index 190
10.11.4 FishEyeEstimator 192
10.12 v.5.6.0 192
10.12.1 Vector2 192
10.12.2 BlackWhiteEstimator 193
10.13 v.5.5.0 193
10.13.0.1 Examples of code 193
10.14 v.5.2.0 194
10.15 v.5.1.0 194
10.16 v.5.0.0 194
10.16.1 Objects creation 194
10.16.1.1 Examples of code 195

10.16.2 Interface of ILicense . . . 195
10.16.2.1 Examples of code . . . 195
10.16.3 Interface of HumanLandmark . . . 197
10.16.3.1 HumanDetectionType . . . 197
10.16.3.2 HumanLandmarks17 . . . 198
10.16.3.3 IHumanLandmarksDetector . . . 198
10.16.4 Interface of IDescriptorBatch . . . 198
10.16.5 Interface of Detection . . . 198
10.16.6 Interface of IDetector . . . 199
10.16.7 IFaceDetectionBatch . . . 199
10.16.8 Interface of IHumanDetector . . . 200
10.16.9 IHumanDetectionBatch . . . 201
10.16.10 Interface of ILivenessFlyingFaces . . . 202
10.17 v.3.10.1 . . . 203
10.17.1 Detector FaceDetV3 changes . . . 203
10.17.2 Detector FaceDetV1, FaceDetV2 changes . . . 203

11 Best practices **204**
11.1 Thread pools . . . 204
11.2 Estimator creation and inference . . . 204
11.3 Using CPU and GPU models for network inference . . . 204
11.3.1 CPU recommendations . . . 204
11.3.2 GPU recommendations . . . 205
11.4 Forking process . . . 205
11.5 Liveness estimator combination . . . 206
11.5.1 Changing the threshold . . . 206
11.5.2 Aggregating the scores . . . 206
11.5.3 Recommended thresholds . . . 206
11.5.4 Possible LivenessOneShotRGBEstimator model combinations . . . 206

12 Device-specific constraints **208**
12.1 Image constraints . . . 208

13 Collecting information for Technical Support **209**
13.1 Contact Technical Support . . . 209
13.2 Specific error . . . 209
13.3 Non-specific error . . . 210
13.4 Unexpected Result . . . 210

14 Useful tools **212**
14.1 Performance testing . . . 212
14.1.1 Key concepts in performance testing . . . 212
14.2 Metrics for performance analysis . . . 212
14.2.1 Common metrics . . . 212
14.2.1.1 Practical use . . . 213
14.3 Performance test parameters . . . 213
14.3.1 Test-specific parameters . . . 213
14.3.2 Batch and sensor parameters . . . 214
14.3.3 iOS-specific parameters . . . 214
14.3.4 Stopping condition parameters . . . 214
14.3.5 Recommendations for parameter selection . . . 215
14.4 Stopping conditions . . . 215
14.4.1 Normal stopping conditions . . . 215
14.4.2 Emergency stopping conditions . . . 216
14.4.2.1 Configuration of emergency stop conditions . . . 216
14.4.3 Special cases . . . 216
14.5 Example console report . . . 217
14.5.1 Structure of the first table . . . 217
14.5.2 Column contents . . . 218
14.5.3 Additional metrics . . . 218
14.5.4 Zero and last iterations . . . 218
14.5.5 Color coding . . . 218
14.5.6 Reasons for stopping . . . 218
14.5.7 Operational vs. final statistics . . . 218
14.6 Performance test challenges . . . 219
14.6.1 Measurement range limitations . . . 219
14.6.2 High-frequency noise . . . 219
14.6.3 Low-frequency noise . . . 219
14.6.4 Test execution duration . . . 219
14.6.5 Artificial constraints efficiency . . . 220
14.6.6 Launch recommendations . . . 220
14.7 Potential improvements . . . 220
14.8 Practical recommendations . . . 220

15 Appendix A. Specifications **221**
15.1 Classification performance . . . 221
15.2 Runtime performance for CentOS Linux environment . . . 221
15.2.1 CPU performance . . . 222
15.2.1.1 CPU. Detector performance . . . 222

15.2.1.2 CPU. HumanDetector performance 223
15.2.1.3 CPU. HumanFaceDetector performance 224
15.2.1.4 CPU. HeadDetector performance 226
15.2.1.5 CPU. Estimations performance with batch interface 227
15.2.1.6 CPU. Estimations performance without batch interface 232
15.2.1.7 CPU. Extractor performance 233
15.2.1.8 CPU. Matcher performance 234
15.2.1.9 CPU. CrowdEstimator performance 235
15.2.2 GPU performance 236
15.2.2.1 GPU. Detector performance 236
15.2.2.2 GPU. HumanDetector performance 236
15.2.2.3 GPU. HeadDetector performance 237
15.2.2.4 GPU. HumanFace detector performance 238
15.2.2.5 GPU. Estimations performance with batch interface 239
15.2.2.6 GPU. Estimations performance without batch interface 244
15.2.2.7 GPU. Extractor performance 245
15.2.2.8 GPU. CrowdEstimator performance 246
15.2.3 Rockchip (Ubuntu 24.04 LTS) 247
15.2.3.1 Rockchip (power) environment. Detector performance 248
15.2.3.2 Rockchip (power) environment. Extractor performance 248
15.2.3.3 Rockchip (power) environment. HeadDetector performance 249
15.2.3.4 Rockchip (power) environment. HumanDetector performance 249
15.2.3.5 Rockchip (power) Estimations performance without batch interface . . 251
15.2.3.6 Rockchip (power) environment. Estimations performance with batch interface 251
15.2.3.7 Rockchip (weak) environment. Detector performance 255
15.2.3.8 Rockchip (weak) environment. Extractor performance 256
15.2.3.9 Rockchip (weak) environment. HeadDetector performance 256
15.2.3.10 Rockchip (weak) environment. HumanDetector performance 257
15.2.3.11 Rockchip (weak) Estimations performance without batch interface . . . 259
15.2.3.12 Rockchip (weak) environment. Estimations performance with batch interface 259
15.3 Runtime performance for embedded environment 263
15.4 Descriptor size 264

16 Appendix B. Glossary **265**
16.1 Descriptor 265
16.2 Cooperative Photoshooting and Recognition 266
16.3 Matching 266

17 Appendix C. FAQ **267**

18 Appendix D. Known issues **268**

18.1 Overall known issues . . . 268

18.1.1 Warnings during the compilation of user code that utilizes the SDK libraries . . . 268

18.1.2 Premature end of JPEG file . . . 268

18.1.3 SDK stuck when run sdk algorithm in separate process after root FaceEngine object initialized . . . 268

18.1.4 Undefined behaviour with multithreaded usage of the FaceEngine and algorithms 269

18.1.5 Floating point exceptions when working with images that have GPU memory residence . . . 269

18.1.6 Coordinate differences for batched detections . . . 270

18.2 CentOS 8 known issues . . . 270

18.2.1 Archive unpacking . . . 270

18.3 Astra Linux known issues . . . 271

18.3.1 Startup error . . . 271

Introduction

This short guide describes core concepts of the product, shows main FaceEngine features and suggests usage scenarios.

This document is not a full-featured API reference manual nor a step by step tutorial. For reference pages, please see Doxygen API documentation that is shipped with FaceEngine. For complete examples, please head to our developer portal.

What this book does, however, is this:

- It describes ideas behind resource management and gives a clue why one or another decision was made. With this in mind, you are ready to write efficient code with FaceEngine;
- It breaks down full face analysis and recognition pipeline in parts and shows how one part affects all the others. This information will help you to adapt FaceEngine to your needs, which is somewhat more productive than blindly following tutorials;
- It details things that are important and omits things that are obvious, so you get information that matters most.

This book is split up into several chapters. There are chapters dedicated to each FaceEngine facility; there are chapters with conceptual overviews; there are chapters with generic information. We tried to write the book starting from low-level concepts and moving on to face detection, description and recognition tasks solving one problem at a time. Although sometimes we just had to give references to chapters ahead, we tried to minimize such jumps.

The opening chapter of this book is called "Core concepts". It will tell you about memory management techniques, object creation and destruction strategies that are widely used across the entire FaceEngine. The following chapters catch up telling how higher level FaceEngine components are created from those building blocks.

Editions and Platforms

FaceEngine supports multiple platforms. Supported software and hardware platforms differ depending on editions.

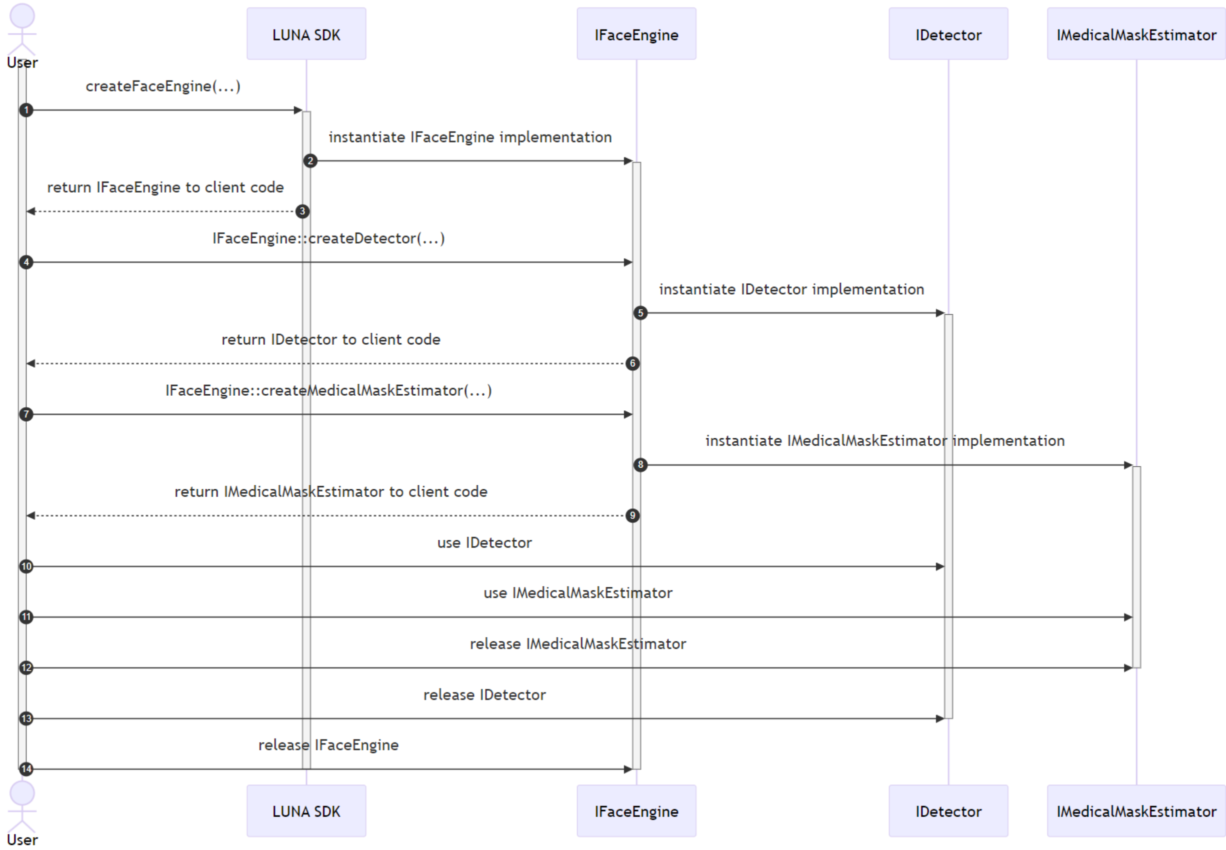
This section includes information about features available for different platforms.

1 Core Concepts

1.1 SDK workflow

1.1.1 Object lifetime

Most of the SDK features are exposed via interfaces (C++ virtual classes) whose implementations must be obtained by calling factory functions. Some of the factories are C-functions, such as `createFaceEngine(...)`. The latter one produces a root object `IFaceEngine`, which in turn exposes many other factories of the `IFaceEngine::createXYZ(...)` form. A typical workflow consists of obtaining `IFaceEngine`, then calling its factories and using the produced child objects.



You do not destroy SDK objects directly, but instead deal with `fsdk::Ref<T>`, reference-counted smart pointers (see section "Automatic reference counting") to SDK interfaces. You only need to release all shared references, at which point `fsdk::Ref<T>` destroys the underlying object.

In terms of lifetime, `IFaceEngine` should outlast all its child objects.

Holding `fsdk::Ref<T>` objects in global variables is error-prone. If the variables are in different translation units, their construction order is undefined, which means the destruction order is out of control, too. Viable approaches include gathering all `fsdk::Ref<T>` objects in a single class or using an explicit stack to store them, as well as storing all `fsdk::Ref<T>` as local variables on the call stack in simple projects. In the case when it is necessary to store `fsdk::Ref<T>` objects as global or static variables, the correct order of releases should be guaranteed explicitly before the program ends:

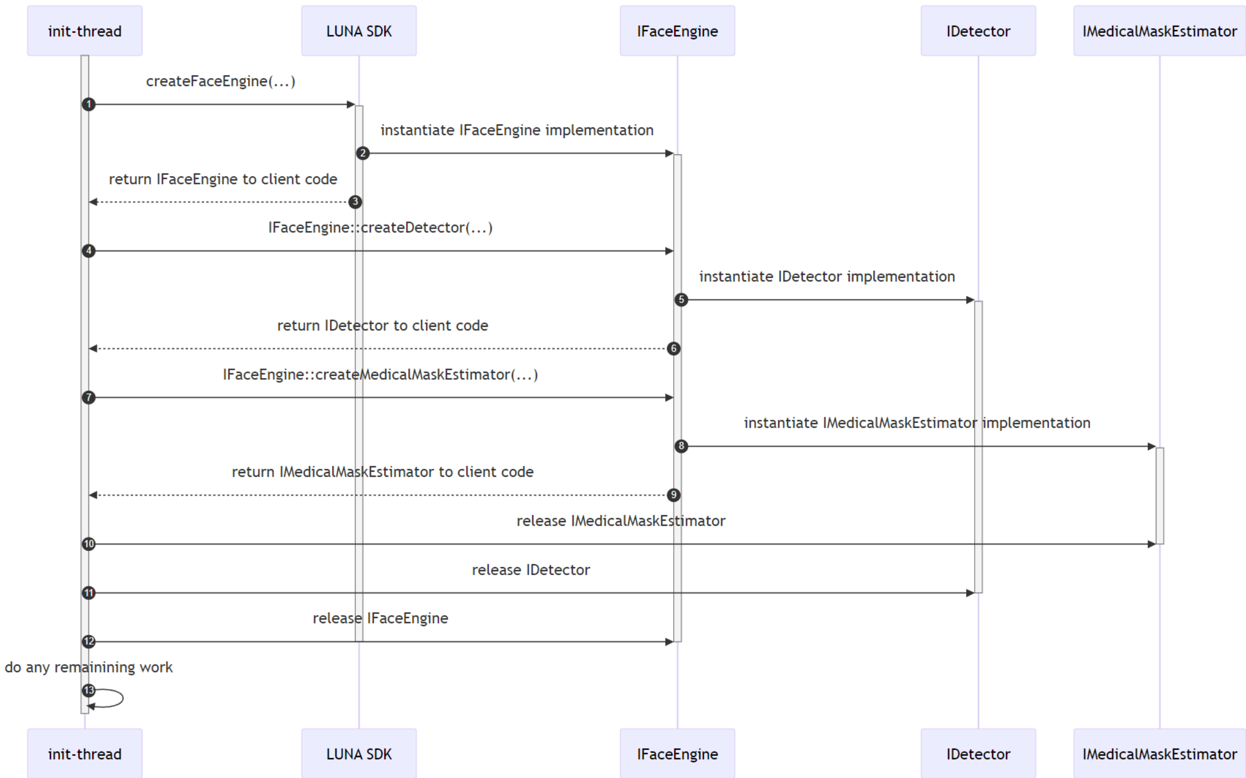
```
//warning: a correct, but not a good example due to these global variables
fsdk::IFaceEnginePtr faceEngine = fsdk::createFaceEngine("./data");
fsdk::IDetectorPtr detector = faceEngine->createDetector();
fsdk::IBestShotQualityEstimator bestShotQualityEstimator = faceEngine->
    createBestShotQualityEstimator();

int main() {
    // application code here

    bestShotQualityEstimator.reset();
    detector.reset();
    faceEngine.reset();
    return 0;
}
```

1.1.2 Threading

The part of the SDK that instantiates and destroys objects is not thread-safe. The SDK requires using one thread (let's call it `init-thread`) for calling all factory functions, as well as releasing the references to the produced objects. The SDK internally uses thread-local objects attached to `init-thread`, which makes `init-thread` special: as long as the SDK is alive, `init-thread` must be alive too. Therefore, there is a requirement that `init-thread` must outlast `IFaceEngine`.



Once SDK objects (such as detectors and estimators, but not `IFaceEngine`) have been created, they are thread-safe and can be used concurrently and on arbitrary threads. Before using an object concurrently on many threads, consider using asynchronous APIs of the SDK instead. For example, `IDetector` along with a synchronous `detect(...)` function also provides asynchronous `detectAsync(...)`.

It is required that an object cannot be destroyed while it has at least one incomplete call, synchronous or asynchronous, on any thread.

1.1.3 Detailed constraints

Here is a more detailed list of lifetime and threading constraints:

- There should be at most one `IFaceEngine` object per process simultaneously. You can create a new `IFaceEngine` object after destroying the previous one, just avoid holding multiple `IFaceEngine` objects at the same time.
- There should be at most one `ITrackEngine` object per process simultaneously. You can create a new `ITrackEngine` object after destroying the previous one, just avoid holding multiple `ITrackEngine` objects at the same time.

> Note: It's not practical to create more than one FE instance from performance standpoint because the same runtime used. You can use it in exceptional cases when settings differ for each instance. In other cases, it's not an error but bug prone behaviour.

- All factory functions should be called on `init-thread` (the thread that calls `createFaceEngine()`). This also implies that factory code is not thread-safe and all factory calls should be serialized in time. Factory functions include:
  - **C-style** functions of the form `createXYZ(...)` such as `createFaceEngine(...)`, `createTrackEngine(...)`
  - **member functions** such as `IFaceEngine::createXYZ(...)`, `ITrackEngine::createXYZ(...)`
- `activateLicense(...)` is not thread-safe. There should be at most one invocation of `activateLicense(...)` per process simultaneously.
- `init-thread` should live no shorter than `IFaceEngine`.
- `IFaceEngine` should live no shorter than `ITrackEngine`.
- `IFaceEngine` should live no shorter than its child objects (algorithms/estimators/detectors). I.e., `IFaceEngine` should be the last destroyed SDK object.
- `IFaceEngine` should be destroyed on `init-thread`.
- Algorithms/estimators/detectors should be destroyed on `init-thread`.
- Algorithms/estimators/detectors can be destroyed when there are no pending or unfinished invocations of member functions of those objects, synchronous or asynchronous, on any threads.
- Track Engine requirements: all Track Engine streams should be stopped, then destroyed, then `ITrackEngine` itself should be stopped, then destroyed.
- `ITrackEngine` and all its streams should be destroyed on `init-thread`.

> Note: Violation of some described constraints may not cause problems right away but in special complex scenario and as program work time passed.

The only part of the SDK that allows multithreading is using member functions of already instantiated algorithms/estimators/detectors, such as `IDetector:detect(...)` and `IAttributeEstimator::estimate(...)`. The member functions can be called on arbitrary threads and in parallel. Before resorting to this multithreaded scenario, please consider using asynchronous versions that accompany many synchronous functions of the SDK.

1.2 Common Interfaces and Types

1.2.1 Reference Counted Interface

Everything in FaceEngine object system starts from here. The *IRefCounted* interface provides methods for reference counter access, increment, and decrement. All reference counted objects imply a custom memory management model. This way they support automated destruction when reference count drops to zero as well as more sophisticated strategies of partial destruction and weak referencing required for FaceEngine internal needs. The bare minimum of such functions is exposed to a user allowing:

- To notify the object that it is required by a client via *retaining* a reference to it.
- To notify the object that it is no longer required by *releasing* a reference to it.
- To get actual reference counter value.

> Reference counted objects expect some special treatment as well. **Be sure never to call *delete* on any pointer to object derived from IRefCounted! Doing so leads to heap corruption.** Simply calling release notifies the system when the object should be destroyed and it does this properly for you.

However, we do not recommend that you interact with the reference counting mechanism manually as doing so may be error-prone. Instead, we recommend that you use smart pointers that are specially designed to handle such objects and provided by FaceEngine. See section "Automatic reference counting" for details.

1.2.2 Automatic reference counting

For your convenience, a special smart pointer class Ref is provided. It is capable of automatic reference counter incrementing upon its creation and automatic decrementing upon its destruction. It also does an assertion of the inner raw pointer being non-null, thus preventing errors.

Two ways of working with Ref are possible:

1.2.2.1 Referencing - without acquiring ownership of object lifetime

```
ISomeObject* createSomeObject();
{
/* Here createSomeObject returns an object with initial reference count of 1
     (otherwise, it would be dead). Then Ref adds another one for itself
    making a total reference count of 2!
*/
Ref<ISomeObject> objref = make_ref(createSomeObject());
/* Here we use the object in any way we want expecting it to be properly
    destroyed when control will leave this scope.
*/

}
/* Here we have left the scope and Ref was automatically destroyed like any
    other object created on the stack. At the same time, it decreased
    reference count of its internal object by 1 making it 1 again.
*/
```

However, the object is not destroyed automatically! For this to happen, it should have precisely 0 references. Moreover, in this example, the raw pointer to the object is lost, so it is impossible to fix it in any way; thus a memory leak is introduced.

1.2.2.2 Acquiring - own object lifetime

So keeping that in mind we introduce a concept of ownership acquiring. By acquiring an object, you mean that its raw pointer is not going to be used and only a valid Ref to it is required. To acquire ownership, use a special ::*acquire()* function. The fixed version of the above example would look like this:

```
ISomeObject* createSomeObject();
{
/* Here createSomeObject returns an object with initial reference count of 1
     (otherwise, it would be dead). Then we acquire it leaving a total
    reference count of 1.
*/
Ref<ISomeObject> objref = acquire(createSomeObject());
/* Here we use the object in any way we want.
*/
}

/* Here we have left the scope and Ref was automatically destroyed like any
    other object created on the stack. At the same time, it decreased
    reference count of its internal object by 1 making it 0. The object is
    destroyed properly by the object system.
*/
```

> Do not store or use raw pointers to the object when using the *::acquire()* function, as ownership acquiring invalidates them.

> Acquiring way of working with Ref is pretty like standard library shared_ptr own lifetime of the object after it returned by *std::make_shared()*.

You can statically cast object type during acquiring or referencing. To achieve this, use special versions of the *::make_ref_as()* and *::acquire_as()* functions. It is your responsibility to ensure that such a cast is possible.

Please refer to FaceEngine Reference Manual for more details on available convenience methods and functions.

As a side note, be informed that *typedef*s for Ref's to all reference counted types are declared. All of them match the following naming convention: *InterfaceName*Ptr. So, for example, Ref<*IDetector*> is equivalent to *IDetectorPtr*.

1.2.3 Serializable object interface

This interface represents an object. Object's contents may be serialized to some data stream and then read back. Think of this as loading and saving.

To interact with the aforementioned data stream, the serializable object needs a user-provided adapter. Such adapter is called the *archive*. See a detailed explanation of it in section "Archive interface" in chapter "Core facility".

Serializable interfaces: *IDescriptor*, *IDescriptorBatch*.

1.2.4 Auxiliary types

1.2.4.1 Image type

Since FaceEngine is a computer vision library, it is natural for it to implement some image concept. Therefore, an *Image* class exists. It is designed as a reference counted container for raw pixel color data. Reference counting allows a single image to be shared by several objects. However, one should understand, that each *Image* object is holding a reference to some data, so if the data is modified in any way, this affects all other objects holding the same reference. To make a deep copy of an *Image*, one should use the *clone()* method, since assignment operators just make a reference. It is also possible to clip a part of an image into a new image by means of *extract()* method.

Pixel data may be characterized by color channel layout, i.e., a number of color channels and their order. The engine defines a *Format* structure for that. The *Format* determines:

- Number of color channels (e.g., RGB or grayscale);
- Order of color channel (e.g., RGB vs. BGR).

FaceEngine assumes 8 bits (i.e., 1 byte) per color channel and implements 8 BPP grayscale, 24 BPP RGB/BGR and padded 32 BPP formats. Format conversion functions are also provided for convenience; see the *convert()* function family.

The *Image* class supports data range mapping. It is possible to map a subset of bytes in a rectangular area for reading or writing. The mapped pixels are represented by the *SubImage* structure. In contrast to *Image*, *SubImage* is just a data view and is *not* reference counted. You are not supposed to store *SubImage*s longer that it is necessary to complete data modification. See the documentation of the *map()* function family for details.

The supports IO roitines to read/write OOM, JPEG, PNG and TIFF formats via FreeImage library.

The absence of image IO is dictated by the fact that FaceEngine focuses on being lightweight and with the minimum possible number of external dependencies. It is not designed solely with image processing purpose in mind. I.e., one may treat video frames as *Image*s and process them one by one. In this case, an external (possibly proprietary) video codec is required.

1.3 Beta Mode

Some features in LUNA SDK are available just in Beta mode. This is experimental features which may be unstable. If you want use them, you have to activate betaMode param in config (faceengine.conf).

2 FaceEngine Structure Overview

FaceEngine is subdivided into several facilities. Each facility is dedicated to a single function. Below there is a list of all facilities with short descriptions of functionality they provide. Detailed information may be found in corresponding chapters of this handbook.

FaceEngine facility list:

- Core facility. This facility stores shared low-level FaceEngine types and factories. This facility is responsible for normal functioning of all other facilities by providing settings accessors and common interfaces. The core facility also contains the main FaceEngine root object that is used to create instances of all higher level objects;
- Face detection facility. This facility is dedicated to object detection. It contains various object detector implementations and factories;
- Parameter estimation facility. This facility is dedicated to various image parameter estimation, such as blurriness, transformation and so forth. It contains various estimator implementations and factories;
- Descriptor processing facility. This facility is dedicated to descriptor extraction and matching. The descriptor is a set of features, describing an object, invariant to object transformation, size or other parameters. Descriptor matching allows judging with certain probability whether two objects are the same. This facility contains various descriptor extractors and containers as well as factories, required to produce them.

So, each facility is a set of classes dedicated to some common for them problem domain. Facilities are independent of each other, with several exceptions, like that all higher level facilities depend on the core facility. Interfacility dependencies are thoroughly described in corresponding chapters of this handbook. The actual set of facilities may vary depending on particular FaceEngine distributions as facilities may be licensed and shipped separately.

This handbook describes the very complete FaceEngine distribution, assuming all facilities are available. The facilities are listed in order of increasing complexity. Applying functions from these facilities in this order allows creating a complete face detection, analysis, recognition and matching pipeline with a significant degree of flexibility. The following chapters break down such pipeline in details.

3 Core Facility

3.1 Common Interfaces

3.1.1 Face Engine Object

The Face Engine object is a root object of the entire FaceEngine. Everything begins with it, so it is essential to create an instance of it. To create a Face Engine instance call *createFaceEngine* function. Also, you may specify default *dataPath* and *configPath* in *createFaceEngine* parameters.

> If you plan to use GPU acceleration, you should keep in mind CUDA runtime initialization and shutdown. Specifically, CUDA creates global runtime object with implicit lifetime; see http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#initialization.

To prevent FaceEngine runtime and lifetime mismatch, it is recommended to avoid creating a static global instance of FaceEngine, as its destruction order is hard to keep track of and control.

3.1.2 Settings Provider

Settings provider is a special entity that loads settings from various locations. Since settings might be shared among several objects, it is useful to cache them to minimize disk reads and provide a dictionary-like interface for named value lookup.

This is what the provider does. The provider object stands somewhat aside FaceEngine facility structure and is created by a separate factory function *createSettingsProvider*. This function accepts configuration file path as a parameter (see section "Configuration data" for details). By default, the engine holds a single provider instance for all facilities. Think of it as a reference counted config file. This provider is passed by the Face Engine object to each factory it creates. The factory, in turn, can read its configuration data from the object and pass it further to its child objects. In typical scenarios, you should not bother with providers as the engine does everything for you. However, when relying on custom factory creation parameters (see the description in section "Face engine object"), you have to create and supply a provider wherever it is required manually.

3.2 Helper Interfaces

3.2.1 Archive Interface

Archive interface is used to provide serialization functions with a data source. It contains methods primarily for data reading and writing. Note, that *IArchive* is not derived from *IRefCounted*, thus does not imply any special memory management strategies.

A few points to keep in mind when implementing your archive:

- FaceEngine objects that use *IArchive* for serialization purposes do call only *write()* (during saving) or only *read()* (during loading) but never both during the same process unless otherwise is explicitly stated;
- During saving or loading FaceEngine objects are free to write or read their data in chunks; e.g., there may be several sequential calls to *write()* in the scope of a single serialization request. The same is true for *read()*. Basically, *read()* and *write()* should behave pretty much like C *fread()* and *fwrite()* standard library functions.

Any *IArchive* implementation should be aware of these notes.

Since these interface methods are pretty obvious and mostly self-explanatory, we advise you to check out FaceEngine Reference Manual for the details.

3.3 Sensor type

SensorType determines which type of camera sensor is used to perform estimation. Currently two types of SensorType are available: `Visible`, `NIR`. The user can indicate the required type of sensor when creating an object by passing the appropriate parameter.

3.4 Data Paths

3.4.1 Model Data

Various FaceEngine modules may require data files to operate. The files contain various algorithm models and constants used at runtime. All the files are gathered together into a single *data* directory. The data directory location is assumed to reside in:

- /opt/visionlabs/data on Linux
- ./data on Windows

One may override the data directory location by means of *setDataDirectory()* method which is available in *IFaceEngine*. Current data location may be retrieved via *getDataDirectory()* method.

3.4.2 Configuration Data

The configuration file is called *faceengine.conf* and stored in */data* directory by default. ConfigurationGuide.pdf with parameter description and default values is located at */doc* package folder.

At runtime, the configuration file data is managed by a special object that implements *ISettingsProvider* interface (see section "Settings provider"). The provider is instantiated by means of *createSettingsProvider()* function that accepts configuration file location as a parameter or uses aforementioned defaults if not specified.

One may supply a different configuration to any factory object by means of *setSettingsProvider()* method, which is available in each factory object interface, including *IFaceEngine*. Currently, bound settings provider may be retrieved via *getSettingsProvider()* method.

4 Detection facility

4.1 Overview

Object detection facility is responsible for quick and coarse detection tasks, like finding a face in an image.

4.2 Detection structure

The detection structure represents an images-space bounding rectangle of the detected object as well as the detection score.

Detection score is a measure of confidence in the particular object classification result and may be used to pick the most "confident" face of many.

Detection score is the measure of classification confidence and not the source image quality. While the score is related to quality (low-quality data generally results in a lower score), it is not a valid metric to estimate the visual quality of an image.

Special estimators exist to fulfill this task (see section "Image Quality Estimation" in chapter "Parameter estimation facility" for details).

4.3 Face Detection

Object detection is performed by the *IDetector* object. The function of interest is *detect()*. It requires an image to detect on and an area of interest (to virtually crop the image and look for faces only in the given location).

Also, face detector implements *detectAsync()* which allows you to asynchronously detect faces and their parameters on multiple images.

Note: Method *detectAsync()* is experimental, and it's interface may be changed in the future.

Note: Method *detectAsync()* is not marked as noexcept and may throw an exception.

4.3.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.

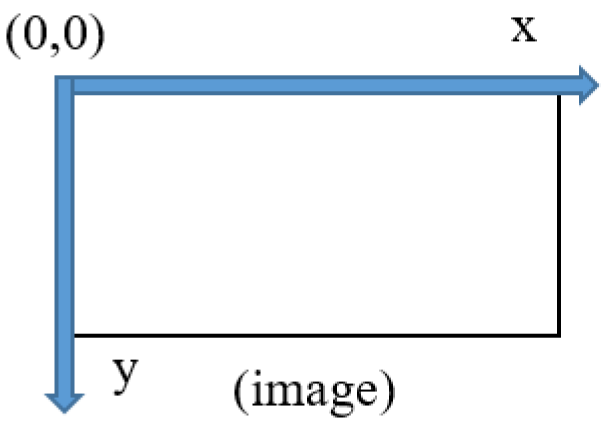


Figure 1: Source image coordinate system

4.3.2 Face detection

When a face is detected, a rectangular area with the face is defined. The area is represented using coordinates in the image coordinate system.

4.3.3 Redetect method

Face detector implements *redetect()* method which is intended for face detection optimization on video frame sequences. Instead of doing full-blown detection on each frame, one may *detect()* new faces at a lower frequency (say, each 5th frame) and just confirm them in between with *redetect()*. This dramatically improves performance at the cost of detection recall. Note that *redetect()* updates face landmarks as well.

Also, face detector implements *redetectAsync()* which allows you to asynchronously redetect faces on multiple images based on the detection results for the previous frames.

Note: Method *redetectAsync()* is experimental, and it's interface may be changed in the future.

Note: Method *redetectAsync()* is not marked as noexcept and may throw an exception.

Detector works faster with larger value of `minFaceSize`.

4.3.4 Orientation Estimation

Name: OrientationEstimator

Algorithm description:

This estimator aims to detect an orientation of the input image. The next outputs are supported:

- The target image is normal oriented ;

- The target image is turned to the left by 90 deg;
- The target image is flipped upside-down;
- The target image is turned to the right by 90 deg.

Implementation description:

The estimator (see IOrientationEstimator in IOrientationEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format and returns the estimation result;
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source images** in R8G8B8 format and `fsdk::Span` of the `fsdk::OrientationType` enums to return results of estimation.

The **OrientationType enumeration** contains all possible results of the Orientation estimation:

```
enum OrientationType : uint32_t {
    OT_NORMAL = 0,      //!< Normal orientation of image
    OT_LEFT = 1,        //!< Image is turned left by 90 deg
    OT_UPSIDE_DOWN = 2, //!< Image is flipped upsidedown
    OT_RIGHT = 3        //!< Image is turned right by 90 deg
};
```

API structure name:

IOrientationEstimator

Plan files:

- orientation_v2_cpu.plan
- orientation_v2_cpu-avx2.plan
- orientation_v2_gpu.plan

4.3.5 Detector variants

Supported detector variants:

- FaceDetV2
- FaceDetV3

There are two basic detector families. The first of them includes FaceDetV2. The second family includes FaceDetV3. FaceDetV3 is the most precise detector. For this type of detector can be passed sensor type.

User code may specify necessary detector type while creating *IDetector* object using parameter.

> FaceDetV2 performance depends on a number of faces in an image and image complexity. FaceDetV3 performance depends only on the target image resolution.

> FaceDetV3 works faster with batched redetect.

> FaceDetV3 supports asynchronous methods for detection and redetection. FaceDetV2 will return a not implemented error.

4.3.6 FaceDetV2 Configuration

FaceDetV2 detector's performance depend on number of faces in image. FaceDetV3 doesn't depend on it.

4.3.7 FaceDetV3 Configuration

FaceDetV3 detects faces from `minFaceSize` to `minFaceSize` * 32. You can change the minimum size of the faces that will be searched in the photo from the `faceengine.conf` configuration.

For example:

```
config->setValue("FaceDetV3::Settings", "minFaceSize", 20);
```

The logic of the detector is very understandable. The smaller the face size we need to find the more time we need.

We recommend to use such meanings for `minFaceSize`: 20, 40 and 90. The size 90 pix is recommended for recognition. If you want to find faces with custom size value you will need to point with size with: `95% * value`. For example we want to find faces with size of 50 pix, it means that in config we should set: `50 * 0.95 ~ 47 pix`.

> FaceDetV3 may provide accurate *5 landmarks* only for faces with sizes greater than 40x40. For smaller faces, it provides less accurate landmarks.

If you have few faces on target images and target face sizes after resize will less then 40x40, it's recommended to require *68 landmarks*.

If you have many faces on target image (greater then 7) it will be faster increase `minFaceSize` to have big enough faces for accurate landmarks estimation.

All last changes in Face Detection logic are described in chapter "Migration guide".

4.3.8 Face Alignment

4.3.8.1 Five landmarks

Face alignment is the process of special key points (called "landmarks") detection on a face. FaceEngine does landmark detection at the same time as the face detection since some of the landmarks are by-products of that detection.

At the very minimum, just **5** landmarks are required: two for eyes, one for a nose tip and two for mouth corners. Using these coordinates, one may warp the source photo image (see Chapter "Image warping") for use with all other FaceEngine algorithms.

All detector may provide *5 landmarks* for each detection without additional computations.

Typical use cases for 5 landmarks:

- Image warping for use with other algorithms:
  - Quality and attribute estimators;
  - Descriptor extraction.

4.3.8.2 Sixty-eight landmarks

More advanced **68-points** face alignment is also implemented. Use this when you need precise information about face and its parts. The detected points look like in the image below.

The *68 landmarks* require additional computation time, so don't use it if you don't need precise information about a face. If you use *68 landmarks* , *5 landmarks* will be reassigned to more precise subset of *68 landmarks*.

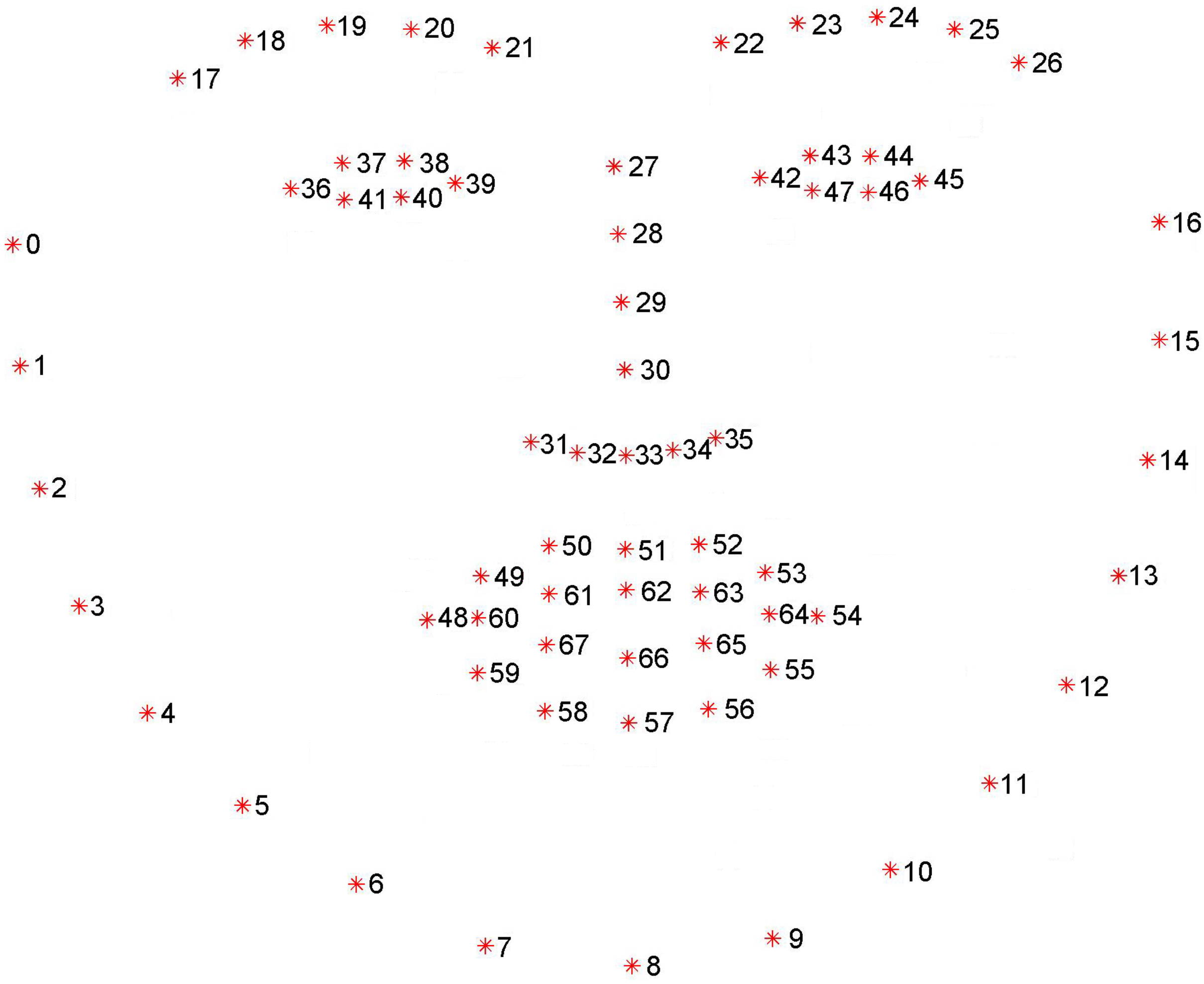


Figure 2: 68-point face alignment

The typical error for landmark estimation on a warped image (see Chapter "Image warping") is in the table below.

Table 1: "Average point estimation error per landmark"

Point	Error (pixels)	Point	Error (pixels)	Point	Error (pixels)	Point	Error (pixels)
1	±3,88	18	±3,77	35	±1,62	52	±1,65
2	±3,53	19	±2,83	36	±1,90	53	±2,01
3	±3,88	20	±2,70	37	±1,78	54	±2,00
4	±4,30	21	±3,06	38	±1,69	55	±1,93
5	±4,67	22	±3,92	39	±1,63	56	±2,18
6	±4,87	23	±3,46	40	±1,52	57	±2,17
7	±4,67	24	±2,59	41	±1,54	58	±1,99
8	±4,01	25	±2,53	42	±1,60	59	±2,32
9	±3,46	26	±2,95	43	±1,55	60	±2,33
10	±3,87	27	±3,84	44	±1,60	61	±2,06
11	±4,56	28	±1,88	45	±1,74	62	±1,97
12	±4,94	29	±1,75	46	±1,72	63	±1,56
13	±4,55	30	±1,92	47	±1,68	64	±1,86
14	±4,45	31	±2,20	48	±1,65	65	±1,94
15	±4,13	32	±1,97	49	±1,99	66	±2,00
16	±3,68	33	±1,70	50	±1,99	67	±1,70
17	±4,09	34	±1,73	51	±1,95	68	±2,12

Simple 5-point landmarks roughly correspond to:

- Average of positions 37, 40 for a left eye;
- Average of positions 43, 46 for a right eye;
- Number 31 for a nose tip;
- Numbers 49 and 55 for mouth corners.

The landmarks for both cases are output by the face detector via Landmarks5 and Landmarks68 structures. Note, that performance-wise 5-point alignment result comes free with a face detection, whereas 68-point result does not. So you should generally request the lowest number of points for your task.

Typical use cases for 68 landmarks:

- Segmentation;
- Head pose estimation.

4.4 Face Landmarks Detector

Every kind of detector provides an interface to find face landmarks. If you have a face detection without landmarks we provide additional interface to request them. The detection of landmarks is performed by the *IFaceLandmarksDetector* object. The functions of interest are *detectLandmarks5()* and *detectLandmarks68*. They need images and detections.

4.5 Human Detection

This functionality enables you to detect human bodies in an image.

Human body detection is performed by the `IHumanDetector` object. The function of interest is `detect()`. It requires an image to detect on.

Also, `IHumanDetector` implements `detectAsync()` which allows you to asynchronously detect human body parameters on multiple images.

> **Note:** Method `detectAsync()` is experimental, and its interface may be changed in the future.

> **Note:** Method `detectAsync()` is not marked as *noexcept* and may throw an exception.

4.5.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.

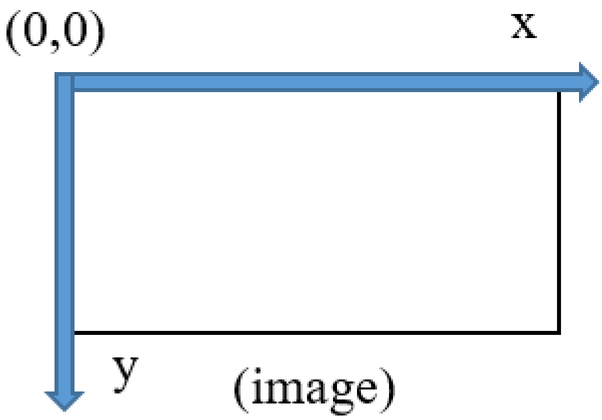


Figure 3: Source image coordinate system

4.5.2 Human body detection

When a human body is detected, a rectangular area with the body is defined. The area is represented using coordinates in the image coordinate system.

4.5.3 Constraints

Human body detection has the following constraints:

- Human body detector works correctly only with adult humans in an image.

- The detector may detect a body of size from 60 px to 640 px (in an image with a long side of 640 px). You can change the input image size in the config. For details, see HumanDetector settings. The image will be resized to the specified size by the larger side while maintaining the aspect ratio.

4.5.4 Camera position requirements

In general, you should locate the camera for human detection according to the image below.

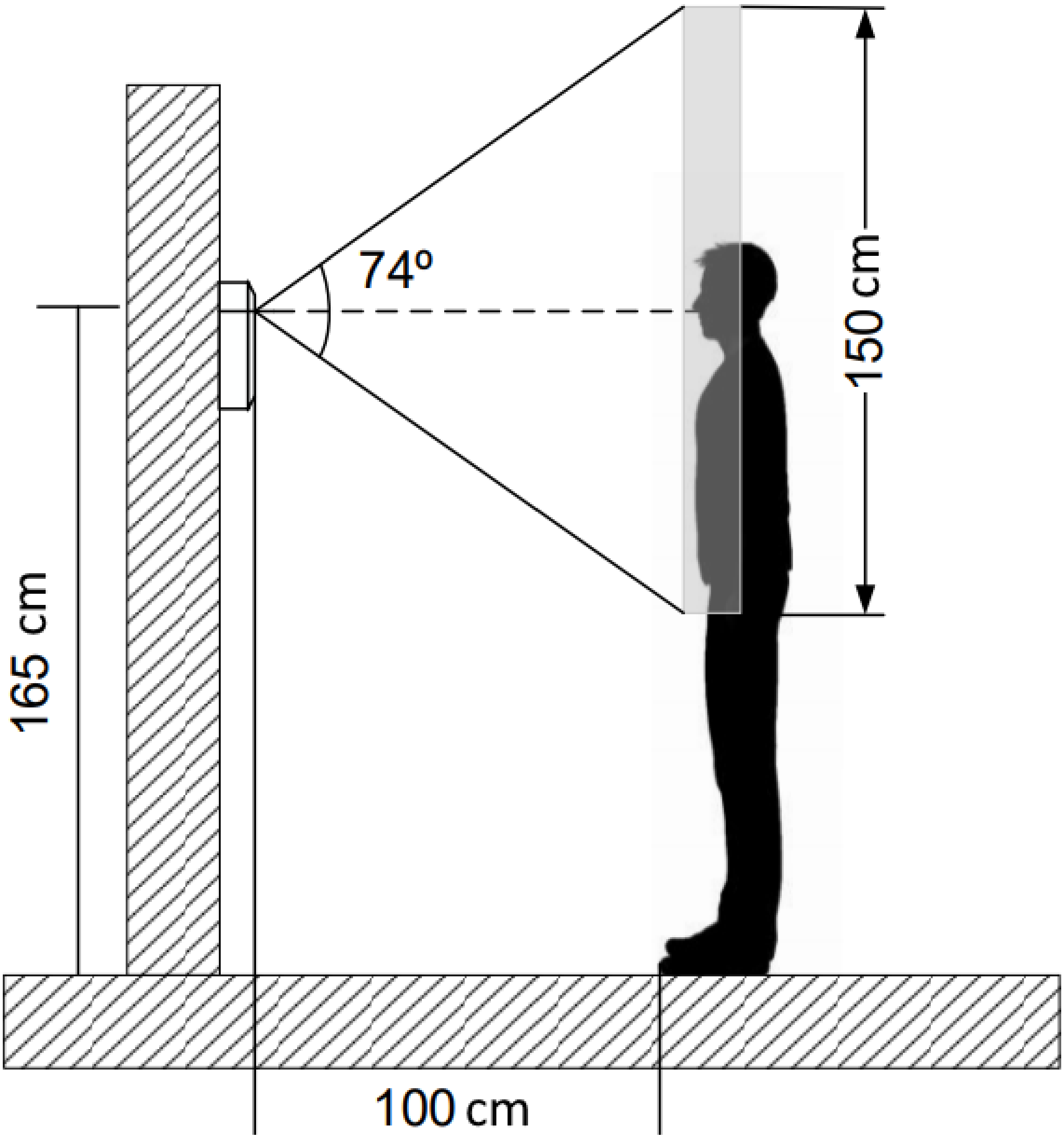


Figure 4: Camera position for human detection

Follow these recommendations to correctly detect human body and keypoints:

- A person's body should face the camera.
- Keep angle of view as close to horizontal as possible.
- There should be about 60% of the person's body in the frame (upper body).
- There must not be any objects that overlap the person's body in the frame.
- The camera tilt angle is recommended from 0 (parallel to the ground) to 60 degrees.

The examples of wrong camera positions are shown in the image below.

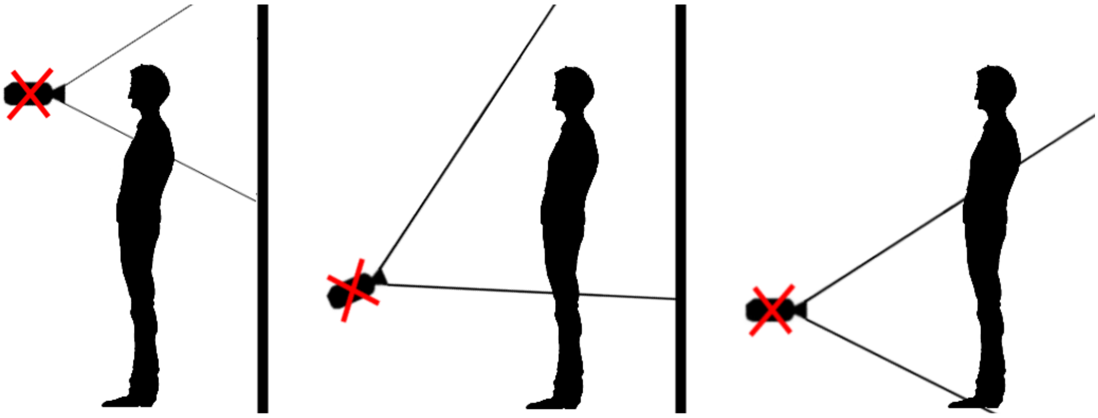


Figure 5: Wrong camera positions

4.5.5 Human body redetection

Like any other detector in Face Engine SDK, human detector also implements redetection model. You can make full detection only in a first frame, and then redetect the same human in the next "n" frames thereby boosting performance of the whole image processing loop.

You can use the `redetectOne()` method, if only a single human detection is required. For more complex use cases, use `redetect()` to redetect humans from multiple images.

Also, `IHumanDetector` implements `redetectAsync()` which allows you to asynchronously redetect human body parameters on multiple images.

> **Note:** Method `redetectAsync()` is experimental, and its interface may be changed in the future.

> **Note:** Method `redetectAsync()` is not marked as *noexcept* and may throw an exception.

4.5.6 Human keypoints

The detector gives an opportunity to detect human body *keypoints* in an image.

The image below shows the keypoints detected for a human body.

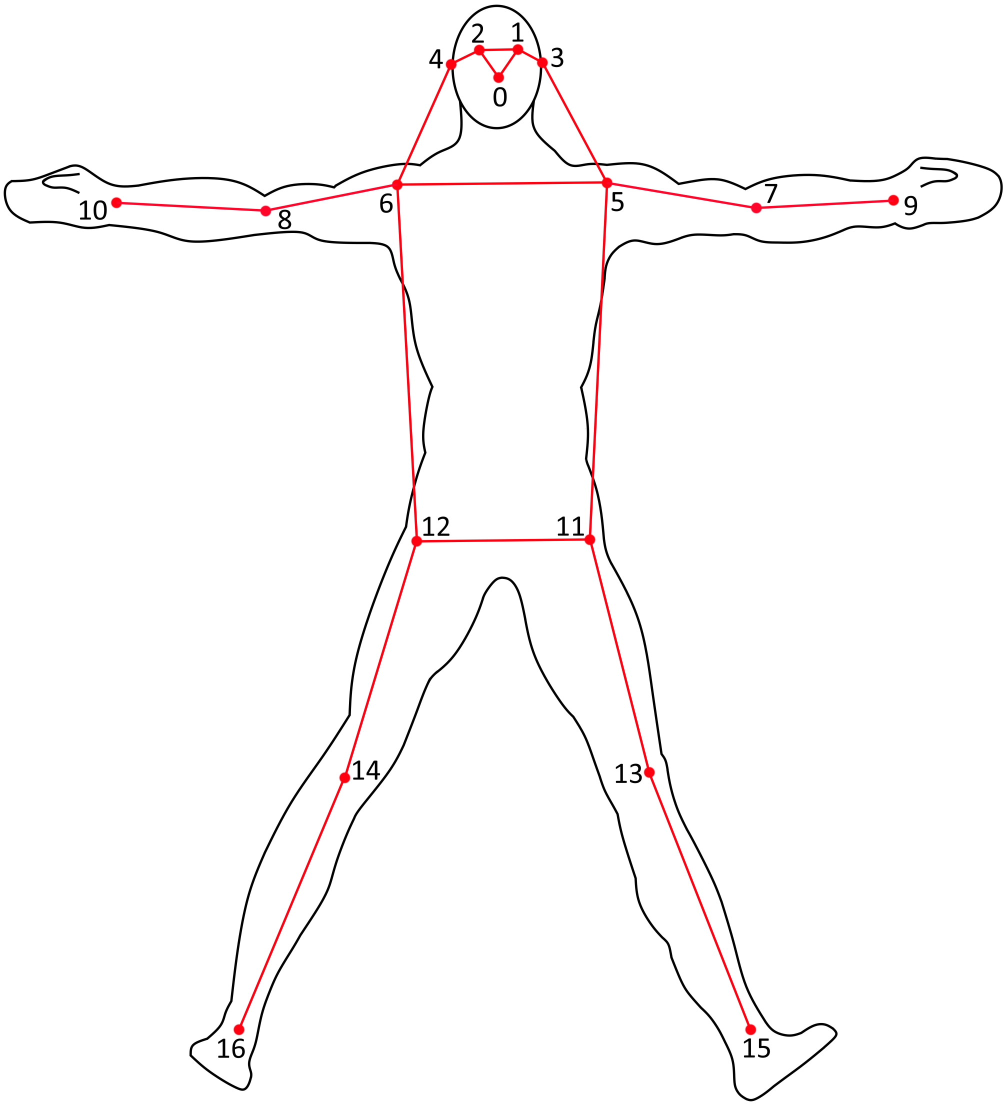


Figure 6: 17-points of human body

Point	Body Part	Point	Body Part
0	Nose	9	Left Wrist

Point	Body Part	Point	Body Part
1	Left Eye	10	Right Wrist
2	Right Eye	11	Left Hip
3	Left Ear	12	Right Hip
4	Right Ear	13	Left Knee
5	Left Shoulder	14	Right Knee
6	Right Shoulder	15	Left Ankle
7	Left Elbow	16	Right Ankle
8	Right Elbow		

Cases that increase the probability of error:

- Non-standard poses (head below the shoulders, vertical twine, lying head to the camera, and so on).
- Camera position from above at a large angle.
- Sometimes estimator predicts invisible points with high score, especially for points of elbows, wrists, ears.

4.5.7 Main results of each detection

The main result of each detection is an array. Each array element consists of a point (`fsdk:: Point2f`) and a score. If the score value is less than the threshold, then the value of "x" and "y" coordinates will be equal to 0.

For more information about thresholds and other configuration parameters, see the HumanDetector settings section oof Configuration Guide. ## HumanFace Detection. Face to body association {#humanface-detection}

This functionality enables you to detect the bodies and faces of people and perform an association between them, determining whether the detected face and body belong to the same person.

This detector contains the implementation of both Human and Face(FaceDetV3) detectors. This means that all the requirements, constraints and recommendations for quality improvement imposed for these detectors will be relevant for the HumanFace detector.

Detector operation algorithm:

- human detection
- face detection
- determination of an association for each detection

Figure 7: HumanFace detection

4.5.8 HumanFace redetection

To perform redetection, you need to separately redetect body and face.

4.5.9 Performance

User can skip computation of associations by selecting according `HumanFaceDetectionType` for `detect()` method, if he doesn't need this functionality. In such case, we estimate performance gain about 5% on cpu and about 20% on gpu devices. The more faces and bodies represented in image, the more gain user will enjoy after association skip.

4.5.10 Main results

There are two output structures:

- **HumanFaceBatch**
- **HumanFaceAssociations**

The **HumanFaceBatch** contains three arrays - face detections, human detections and associations:

```
struct IHumanFaceBatch : public IRefCounted {
    virtual Span<const Detection> getHumanDetections(size_t index = 0)
        const noexcept = 0;
    virtual Span<const Detection> getFaceDetections(size_t index = 0)
        const noexcept = 0;
```

```
        virtual Span<const HumanFaceAssociation> getAssociations(size_t
            index = 0) const noexcept = 0;
    };
```

The **HumanFaceAssociation structure** contains results of the association:

```
    struct HumanFaceAssociation {
        uint32_t humanId;
        uint32_t faceId;
        float score;
    };
```

There are two groups of fields:

1⊠ The first group contains body and face detection indexes:

```
        uint32_t humanId;
        uint32_t faceId;
```

2⊠ The second group contains association score:

```
        float score;
```

The score is defined in [0,1] range.

Associations and detections whose scores are lower than the threshold will be rejected and not returned in the results.

> See ConfigurationGuide.pdf ("HumanFace settings" section) for more information about thresholds and configuration parameters.

4.5.11 minFaceSize

This detector could detect faces with size 20 px and more (`minFaceSize` parameter) and humans with size 100 px and more. In case if such small faces and humans are not required, user could change the `minFaceSize` parameter in the config.

Before processing, the images will be resized by `minFaceSize/20` times. For example, if the value is `minFaceSize=50`, then the image will be additionally resized by `minFaceSize=50/20=2.5` times.

Detector works faster with larger value of `minFaceSize`.

4.6 Head Detection

This functionality enables you to detect the heads of people.

This detector implementation is similar to Face(FaceDetV3) detectors. This means that all the requirements, constraints and recommendations for quality improvement imposed for this detector will be relevant for the Head detector.

Object detection is performed by the *IHeadDetector*. The function of interest is *detect()*. It requires an image to detect on and an area of interest (to virtually crop the image and look for heads only in the given location).

4.6.1 Image coordinate system

The origin of the coordinate system for each processed image is located in the upper left corner.

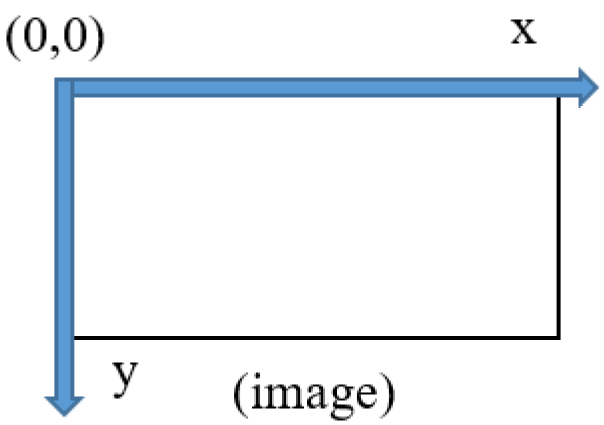


Figure 8: Source image coordinate system

4.6.2 Main results

Output structures:

- **DetectionBatch**

The **DetectionBatch** contains an array of head detections:

```
    struct IDetectionBatch : public IRefCounted {

        virtual size_t getSize() const noexcept = 0;
```

```
        virtual Span<const Detection> getDetections(size_t index = 0) const
            noexcept = 0;

    };
```

4.6.3 minHeadSize

This detector could detect heads with size 20 px and more (`minHeadSize` parameter). In case if such small heads, user could change the `minHeadSize` parameter in the config.

Before processing, the images will be resized by `minHeadSize/20` times. For example, if the value is `minHeadSize=50`, then the image will be additionally resized by `minHeadSize=50/20=2.5` times.

Detector works faster with larger value of `minHeadSize`.

5 Image Warping

Warping is the process of face image normalization. It requires landmarks and face detection (see chapter "Detection facility") to operate. The purpose of the process is to:

- compensate image plane rotation (roll angle);
- center the image using eye positions;
- properly crop the image.

This way all warped images look the same and one can tell that, e.g., left eye is always in a box, defined by the certain coordinates. This way certain transform invariance is achieved for input data so various algorithms can perform better.

The warper (see IWarper in IWarper.h):

- Implements the *warp()* function that accepts span of source `fsdk::Image` in R8B8G8 format, span of `fsdk::Transformation` and span of output `fsdk::Image` structures;
- Implements the *warpAsync()* function that accepts span of source `fsdk::Image` in R8B8G8 format and span of `fsdk::Transformation`.

Note: Method *warpAsync()* is experimental, and it's interface may be changed in the future. **Note:** Method *warpAsync()* is not marked as noexcept and may throw an exception.

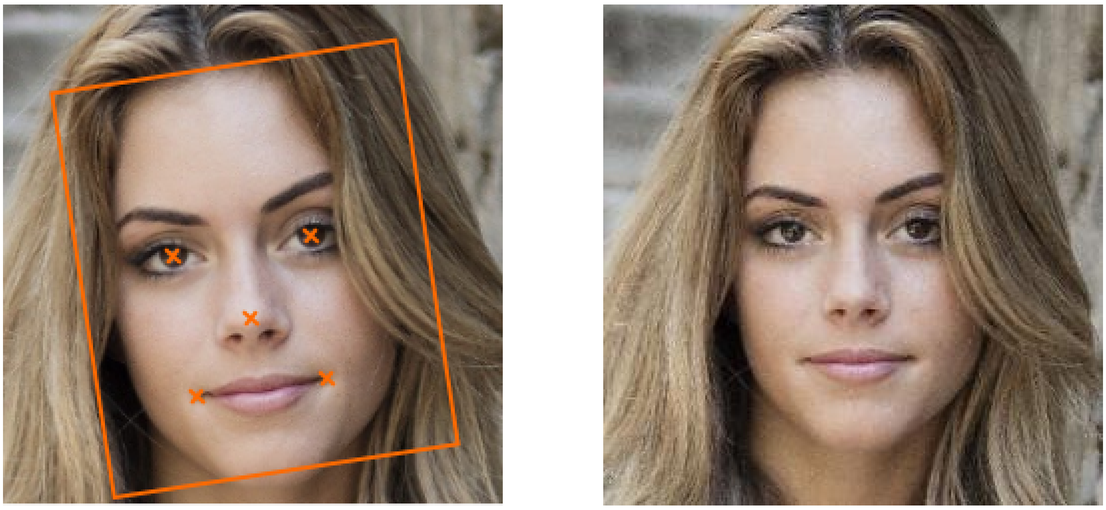


Figure 9: Face warping

Be aware that image warping is not thread-safe, so you have to create a *warper* object per worker thread.

6 Parameter Estimation Facility

6.1 Overview

The estimation facility is the only multi-purpose facility in FaceEngine. It is designed as a collection of tools that help to estimate various images or depicted object properties. These properties may be used to increase the precision of algorithms implemented by other FaceEngine facilities or to accomplish custom user tasks.

6.2 Use cases

6.2.1 ISO estimation

LUNA SDK provides algorithms for image check according to the requirements of the ISO/IEC 19794-5:2011 standard and compatible standards.

The requirements can be found on the official website: https://www.iso.org/obp/ui/#iso:std:iso-iec:19794:-5:en.

The following algorithms are provided:

- Head rotation angles (pitch, yaw, and roll angles). According to section "7.2.2 Pose" in the standard, the angles should be +/- 5 degrees from frontal in pitch and yaw, less than +/- 8 degrees from frontal in roll. See additional information about the algorithm in section "Head Pose".
- Gaze. See section "7.2.3 Expression" point "e" of the standard. See additional information about the algorithm in section "Gaze Estimation".
- Mouth state (opened, closed, occluded) and additional properties for smile (regular smile, smile with teeths exposed) See section "7.2.3 Expression" points "a", "b", and "c" of the standard. See additional information about the algorithm in section "Mouth Estimation".
- Quality of the image:
  - Contrast and saturation (insufficient or too large exposure). See sections "7.2.7 Subject and scene lighting" and "7.3.2 Contrast and saturation" of the standard.
  - Blurring. See section "7.3.3 Focus and depth of field" of the standard.
  - Specularity. See section "7.2.8 Hot spots and specular reflections" and "7.2.12 Lighting artefacts" of the standard.
  - Uniformity of illumination. See sections "7.2.7 Subject and scene lighting" and "7.2.12 Lighting artefacts" of the standard.

  See additional information about the algorithm in section "Image Quality Estimation".
- Glasses state (no glasses, glasses, sunglasses). See section "7.2.9 Eye glasses" of the standard. See additional information about the algorithm in section "Glasses Estimation".

- Eyes state (for each eye: opened, closed, occluded). See sections “7.2.3 Expression” point “a”, “7.2.11 Visibility of pupils and irises” and “7.2.13 Eye patches” of the standard. See additional information about the algorithm in section “Eyes Estimation”.
- Natural light estimation. See section “7.3.4 Unnatural colour” of the standard. See additional information about the algorithm in section “Natural Light Estimation”.
- Eybrows state: neutral, raised, squinting, frowning. See section “7.2.3 Expression” points “d”, “f”, and “g” of the standard. See additional information about the algorithm in section “Eyebrows estimation”.
- Position of a person’s shoulders in the original image: the shoulders are parallel to the camera or not. See section “7.2.5 Shoulders” of the standard. See additional information about the algorithm in section “Portrait Style Estimation”.
- Headwear. Checks if there is a headwear on a person or not. Several types of headwear can be estimated. See section “B.2.7 Head coverings” of the standard. See additional information about the algorithm in section “Headwear Estimation”.
- Red eyes estimation. Checks if there is a red eyes effect. See section “7.3.4 Unnatural colour” of the standard. See additional information about the algorithm in section “Red Eyes Estimation”.
- Radial distortion estimation. See section “7.3.6 Radial distortion of the camera lens” of the standard. See additional information about the algorithm in section “Fish Eye Estimation”.
- Image type estimation: color, grayscale, infrared. See section “7.4.4 Use of near infra-red cameras” of the standard. See additional information about the algorithm in section “Grayscale, color or infrared Estimation”.
- Background estimation: background uniformity and if a background is too light or too dark. See section “B.2.9 Backgrounds” of the standard. See additional information about the algorithm in section “Background Estimation”.

6.3 Best shot selection functionality

6.3.1 BestShotQuality Estimation

Name: BestShotQualityEstimator

Algorithm description:

The BestShotQuality estimator is designed to evaluate image quality to choose the best image before descriptor extraction. The BestShotQuality estimator consists of two components - AGS (garbage score) and Head Pose.

AGS aims to determine the source image score for further descriptor extraction and matching.

Estimation output is a float score which is normalized in range [0..1]. The closer score to 1, the better matching result is received for the image.

When you have several images of a person, it is better to save the image with the highest AGS score.

> Recommended threshold for AGS score is equal to **0.2**. But it can be changed depending on the purpose of use. Consult VisionLabs about the recommended threshold value for this parameter.

Head Pose determines person head rotation angles in 3D space, namely pitch, yaw and roll.

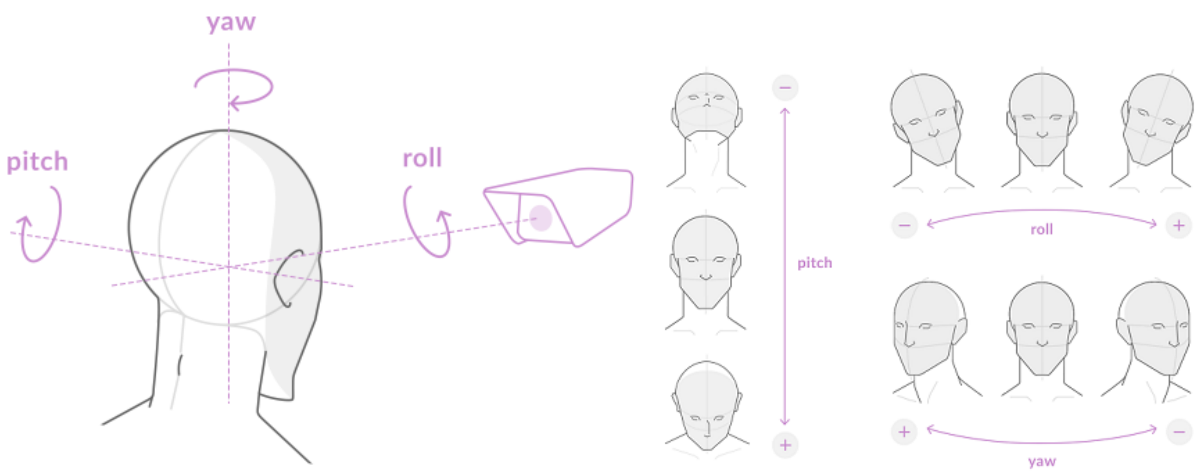


Figure 10: Head pose

Since 3D head translation is hard to determine reliably without camera-specific calibration, only 3D rotation component is estimated.

Head pose estimation characteristics:

- Units (degrees);
- Notation (Euler angles);
- Precision (see table below).

Implementation description:

The estimator (see IBestShotQualityEstimator in IEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Image` in R8G8B8 format, `fsdk::Detection` structure of corresponding **source image** (see section "Detection structure" in chapter "Face detection facility"), `fsdk::IBestShotQualityEstimator::EstimationRequest` structure and `fsdk::IBestShotQualityEstimator::EstimationResult` to store estimation result;
- Implements the *estimate()* function that needs the span of `fsdk::Image` in R8G8B8 format, the span of `fsdk::Detection` structures of corresponding **source images** (see section "Detection

structure" in chapter "Face detection facility"), `fsdk::IBestShotQualityEstimator::EstimationRequest` structure and span of `fsdk::IBestShotQualityEstimator::EstimationResult` to store estimation results.

- Implements the *estimateAsync()* function that needs `fsdk::Image` in R8G8B8 format, `fsdk::Detection` structure of corresponding source image (see section "Detection structure" in chapter "Face detection facility"), `fsdk::IBestShotQualityEstimator::EstimationRequest` structure;

Note: Method *estimateAsync()* is experimental, and it's interface may be changed in the future. **Note:** Method *estimateAsync()* is not marked as noexcept and may throw an exception.

Before using this estimator, user is free to decide whether to estimate or not some listed attributes. For this purpose, *estimate()* method takes one of the estimation requests:

- `fsdk::IBestShotQualityEstimator::EstimationRequest::estimateAGS` to make only AGS estimation;
- `fsdk::IBestShotQualityEstimator::EstimationRequest::estimateHeadPose` to make only Head Pose estimation;
- `fsdk::IBestShotQualityEstimator::EstimationRequest::estimateAll` to make both AGS and Head Pose estimations;

The **EstimationResult structure** contains results of the estimation:

```
struct EstimationResult {
    Optional<HeadPoseEstimation> headPose;  //!< HeadPose estimation if
        was requested, empty otherwise
    Optional<float> ags;                    //!< AGS estimation if was
        requested, empty otherwise
};
```

Head Pose accuracy:

Prediction precision decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table below.

Table 3: "Head pose prediction precision"

	Range	-45°...+45°	< -45° or > +45°
Average prediction error (per axis)	Yaw	±2.7°	±4.6°
Average prediction error (per axis)	Pitch	±3.0°	±4.8°
Average prediction error (per axis)	Roll	±3.0°	±4.6°

Zero position corresponds to a face placed orthogonally to camera direction, with the axis of symmetry parallel to the vertical camera axis.

API structure name:

IBestShotQualityEstimator

Plan files:

For more information see `Approximate Garbage Score Estimation (AGS)` and `Head Pose Estimation`

6.3.2 Image Quality Estimation

Name: QualityEstimator

> **DEPRECATED (since v.5.33.0):** `IQualityEstimator` is deprecated. Use `ISubjectiveQualityEstimator` instead.

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details).

This estimator is designed to determine the image quality. You can estimate the image according to the following criteria:

- The image is blurred;
- The image is underexposed (i.e., too dark);
- The image is overexposed (i.e., too light);
- The face in the image is illuminated unevenly (there is a great difference between light and dark regions);
- Image contains flares on face (too specular).

Examples are presented in the images below. Good quality images are shown on the right.

Figure 11: Blurred image (left), not blurred image (right)

Figure 12: Dark image (left), good quality image (right)

Figure 13: Light image (left), good quality image (right)

Figure 14: Image with uneven illumination (left), image with even illumination (right)

Figure 15: Image with specularity - image contains flares on face (left), good quality image (right)

Implementation description:

The general rule of thumb for quality estimation:

1. Detect a face, see if detection confidence is high enough. If not, reject the detection.
2. Produce a warped face image (see chapter "Descriptor processing facility") using a face detection and its landmarks.

3. Estimate visual quality using the estimator, finally reject low-quality images.

While the scheme above might seem a bit complicated, it is the most efficient performance-wise, since possible rejections on each step reduce workload for the next step.

At the moment estimator exposes two interface functions to predict image quality:

- **virtual Result estimate(const Image& warp, Quality& quality);**
- **virtual Result estimate(const Image& warp, SubjectiveQuality& quality);**

Each one of this functions use its own CNN internally and return slightly different quality criteria.

The first CNN is trained specifically on pre-warped human face images and will produce lower score factors if one of the following conditions are satisfied:

- Image is blurred;
- Image is under-exposured (i.e., too dark);
- Image is over-exposured (i.e., too light);
- Image color variation is low (i.e., image is monochrome or close to monochrome).

Each one of this score factors is defined in [0..1] range, where higher value corresponds to better image quality and vice versa.

The second interface function output will produce lower factor if:

- The image is blurred;
- The image is underexposed (i.e., too dark);
- The image is overexposed (i.e., too light);
- The face in the image is illuminated unevenly (there is a great difference between light and dark regions);
- Image contains flares on face (too specular).

The estimator determines the quality of the image based on each of the aforementioned parameters. For each parameter, the estimator function returns two values: the quality factor and the resulting verdict.

As with the first estimator function the second one will also return the quality factors in the range [0..1], where 0 corresponds to low image quality and 1 to high image quality. E. g., the estimator returns low quality factor for the Blur parameter, if the image is too blurry.

The resulting verdict is a quality output based on the estimated parameter. E. g., if the image is too blurry, the estimator returns "isBlurred = true".

The threshold (see below) can be specified for each of the estimated parameters. The resulting verdict and the quality factor are linked through this threshold. If the received quality factor is lower than the threshold, the image quality is low and the estimator returns "true". E. g., if the image blur quality factor is higher than the threshold, the resulting verdict is "false".

If the estimated value for any of the parameters is lower than the corresponding threshold, the image is considered of bad quality. If resulting verdicts for all the parameters are set to "False" the quality of the image is considered good.

The quality factor is a value in the range [0..1] where 0 corresponds to low quality and 1 to high quality.

> Illumination uniformity corresponds to the face illumination in the image. The lower the difference between light and dark zones of the face, the higher the estimated value. When the illumination is evenly distributed throughout the face, the value is close to "1".

> Specularity is a face possibility to reflect light. The higher the estimated value, the lower the specularity and the better the image quality. If the estimated value is low, there are bright glares on the face.

The **Quality structure** contains results of the estimation made by first CNN. Each estimation is given in normalized [0, 1] range:

```
struct Quality {
    float light;    //!< image overlighting degree. 1 - ok, 0 -
        overlighted.
    float dark;     //!< image darkness degree. 1 - ok, 0 - too dark.
    float gray;     //!< image grayness degree 1 - ok, 0 - too gray.
    float blur;     //!< image blur degree. 1 - ok, 0 - too blured.
    inline float getQuality() const noexcept;   //!< complex estimation
        of quality. 0 - low quality, 1 - high quality.
};
```

The **SubjectiveQuality structure** contains results of the estimation made by second CNN. Each estimation is given in normalized [0, 1] range:

```
struct SubjectiveQuality {
    float blur;         //!< image blur degree. 1 - ok, 0 - too blured.
    float light;        //!< image brightness degree. 1 - ok, 0 - too
        bright;
    float darkness;     //!< image darkness degree. 1 - ok, 0 - too dark
        ;
    float illumination; //!< image illumination uniformity degree. 1 -
        ok, 0 - is too illuminated;
    float specularity;  //!< image specularity degree. 1 - ok, 0 - is
        not specular;
    bool isBlurred;     //!< image is blurred flag;
    bool isHighlighted; //!< image is overlighted flag;
    bool isDark;        //!< image is too dark flag;
    bool isIlluminated; //!< image is too illuminated flag;
    bool isNotSpecular; //!< image is not specular flag;
```

```
        inline bool isGood() const noexcept;    //!< if all boolean flags
           are false returns true - high quality, else false - low quality.
    };
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `QualityEstimator::Settings` section. By default, these threshold values are set to optimal.

Table 4: "Image quality estimator recommended thresholds"

Threshold	Recommended value
blurThreshold	0.58
lightThreshold	0.58
darknessThreshold	0.52
illuminationThreshold	0.3
specularityThreshold	0.3

The most important parameters for face recognition are "blurThreshold", "darknessThreshold" and "lightThreshold", so you should select them carefully.

You can select images of better visual quality by setting higher values of the "illuminationThreshold" and "specularityThreshold". Face recognition is not greatly affected by uneven illumination or glares.

Configurations:

See the "Quality estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IQualityEstimator

Plan files:

- model_subjective_quality_<version>_cpu.plan
- model_subjective_quality_<version>_cpu-avx2.plan
- model_subjective_quality_<version>_gpu.plan

Note: `usePlanV1` toggles the `Quality` estimation, `usePlanV2` toggles the `SubjectiveQuality` estimation. These parameters can enable or disable the corresponding functionality via the faceengine.conf configuration file.

```xml
<section name="QualityEstimator::Settings">
...
    <param name="usePlanV1" type="Value::Int1" x="1" />
    <param name="usePlanV2" type="Value::Int1" x="1" />
</section>
```

Note that you cannot disable both the parameters at the same time. In case you do this, you will receive the `fsdk::FSDKError::InvalidConfig` error code and the following logs:

```
[27.06.2024 12:38:59] [Error] QualityEstimator::Settings Failed to create
    QualityEstimator! The both parameters: "usePlanV1" and "usePlanV2" in
    section "QualityEstimator::Settings" are disabled at the same time.
```

6.3.3 Image Subjective Quality Estimation

Name: SubjectiveQualityEstimator

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details).

This estimator is designed to determine the image quality. You can estimate the image according to the following criteria:

- The image is blurred;
- The image is underexposed (i.e., too dark);
- The image is overexposed (i.e., too light);
- The face in the image is illuminated unevenly (there is a great difference between light and dark regions);
- Image contains flares on face (too specular).

Examples are presented in the images below. Good quality images are shown on the right.

Figure 16: Blurred image (left), not blurred image (right)

Figure 17: Dark image (left), good quality image (right)

Figure 18: Light image (left), good quality image (right)

Figure 19: Image with uneven illumination (left), image with even illumination (right)

Figure 20: Image with specularity - image contains flares on face (left), good quality image (right)

Implementation description:

The general rule of thumb for quality estimation:

1. Detect a face, see if detection confidence is high enough. If not, reject the detection.
2. Produce a warped face image (see chapter "Descriptor processing facility") using a face detection and its landmarks.

3. Estimate visual quality using the estimator, finally reject low-quality images.

While the scheme above might seem a bit complicated, it is the most efficient performance-wise, since possible rejections on each step reduce workload for the next step.

At the moment the estimator exposes an interface function to predict image quality:

- **virtual Result estimate(const Image& warp, SubjectiveQuality& quality);**

This function uses its own CNN internally and returns several subjective quality criteria.

The CNN will produce lower score factors if:

- The image is blurred;
- The image is underexposed (i.e., too dark);
- The image is overexposed (i.e., too light);
- The face in the image is illuminated unevenly (there is a great difference between light and dark regions);
- The image contains flares on face (too specular).

The estimator determines the quality of the image based on each of the aforementioned parameters. For each parameter, the estimator function returns two values: the quality factor and the resulting verdict.

All quality factors are defined in the [0..1] range, where 0 corresponds to low image quality and 1 to high image quality. For example, the estimator returns a low quality factor for the Blur parameter if the image is too blurry.

The resulting verdict is a quality output based on the estimated parameter. For example, if the image is too blurry, the estimator returns "isBlurred = true".

The threshold (see below) can be specified for each of the estimated parameters. The resulting verdict and the quality factor are linked through this threshold. If the received quality factor is lower than the threshold, the image quality is low and the estimator returns "true". For example, if the image blur quality factor is higher than the threshold, the resulting verdict is "false".

If the estimated value for any of the parameters is lower than the corresponding threshold, the image is considered of bad quality. If the resulting verdicts for all the parameters are set to "False", the quality of the image is considered good.

The quality factor is a value in the range [0..1] where 0 corresponds to low quality and 1 to high quality.

> Illumination uniformity corresponds to the face illumination in the image. The lower the difference between light and dark zones of the face, the higher the estimated value. When the illumination is evenly distributed throughout the face, the value is close to "1".

> Specularity is a face possibility to reflect light. The higher the estimated value, the lower the

> specularity and the better the image quality. If the estimated value is low, there are bright glares on the face.

The **SubjectiveQuality structure** contains results of the estimation made by second CNN. Each estimation is given in normalized [0, 1] range:

```
struct SubjectiveQuality {
    float blur;          //!< image blur degree. 1 - ok, 0 - too blured.
    float light;         //!< image brightness degree. 1 - ok, 0 - too
       bright;
    float darkness;      //!< image darkness degree. 1 - ok, 0 - too dark
       ;
    float illumination; //!< image illumination uniformity degree. 1 -
       ok, 0 - is too illuminated;
    float specularity;  //!< image specularity degree. 1 - ok, 0 - is
       not specular;
    bool isBlurred;      //!< image is blurred flag;
    bool isHighlighted; //!< image is overlighted flag;
    bool isDark;         //!< image is too dark flag;
    bool isIlluminated; //!< image is too illuminated flag;
    bool isNotSpecular; //!< image is not specular flag;
    inline bool isGood() const noexcept;    //!< if all boolean flags
       are false returns true - high quality, else false - low quality.
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `SubjectiveQualityEstima` `::Settings` section. By default, these threshold values are set to optimal.

Table 5: "Image quality estimator recommended thresholds"

Threshold	Recommended value
blurThreshold	0.58
lightThreshold	0.58
darknessThreshold	0.52
illuminationThreshold	0.3
specularityThreshold	0.3

The most important parameters for face recognition are "blurThreshold", "darknessThreshold" and “lightThreshold”, so you should select them carefully.

You can select images of better visual quality by setting higher values of the “illuminationThreshold” and “specularityThreshold”. Face recognition is not greatly affected by uneven illumination or glares.

Configurations:

See the “Subjective Quality estimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

ISubjectiveQualityEstimator

Plan files:

- model_subjective_quality_<version>_cpu.plan
- model_subjective_quality_<version>_cpu-avx2.plan
- model_subjective_quality_<version>_gpu.plan

6.4 Attributes estimation functionality

6.4.1 Face Attribute Estimation

Name: AttributeEstimator

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details).

The Attribute estimator determines face attributes. Currently, the following attributes are available:

- Age: determines person's age;
- Gender: determines person's gender;

The Attribute estimator returns `Ethnicity estimation structure`. Each estimation is given in normalized [0, 1] range.

The Ethnicity estimation structure looks like the struct below:

```
struct EthnicityEstimation {
 float africanAmerican;
 float indian;
 float asian;
 float caucasian;

 enum Ethnicities {
  AfricanAmerican = 0,
  Indian,
  Asian,
  Caucasian,
  Count
 };

 /**
  * @brief Returns ethnicity with greatest score.
  * @see EthnicityEstimation::Ethnicities for more info.
  * */
 inline Ethnicities getPredominantEthnicity() const;

 /**
  * @brief Returns score of required ethnicity.
  * @param [in] ethnicity ethnicity.
  * @see EthnicityEstimation::Ethnicities for more info.
  * */
 inline float getEthnicityScore(Ethnicities ethnicity) const;
};
```

Implementation description:

Implementation description:

Before using attribute estimator, user is free to decide whether to estimate or not some specific attributes listed above through *IAttributeEstimator::EstimationRequest* structure, which later get passed in main *estimate()* method. Estimator overrides *IAttributeEstimator::AttributeEstimationResult* output structure, which consists of optional fields describing results of user requested attributes.

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `AttributeEstimator::Settings` section. By default, these threshold values are set to optimal.

Table 6: "Attribute estimator recommended thresholds"

Threshold	Recommended value
genderThreshold	0.5
adultThreshold	0.2

Accuracy:

Age:

- For cooperative (see "Appendix B. Glossary") conditions: average error depends on person age, see table below for additional details. Estimation accuracy is 2.3.

Gender:

- Estimation accuracy in cooperative mode is 99.81% with the threshold 0.5;
- Estimation accuracy in non-cooperative mode is 92.5%.

Table 7: "Average age estimation error per age group for cooperative conditions"

Age (years)	**Average error (years)**
0-3	±3.3
4-7	±2.97
8-12	±3.06
13-17	±4.05
17-20	±3.89
20-25	±1.89

Age (years)	Average error (years)
25-30	±1.88
30-35	±2.42
35-40	±2.65
40-45	±2.78
45-50	±2.88
50-55	±2.85
55-60	±2.86
60-65	±3.24
65-70	±3.85
70-75	±4.38
75-80	±6.79

> In earlier releases of Luna SDK Attribute estimator worked poorly in non-cooperative mode (only 56% gender estimation accuracy), and did not estimate child's age. Having solved these problems average estimation error per age group got a bit higher due to extended network functionality.

Configurations:

See the "AttributeEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IAttributeEstimator

Plan files, precise network type:

- attributes_estimation_precise_<version>_cpu.plan
- attributes_estimation_precise_<version>_cpu-avx2.plan
- attributes_estimation_precise_<version>_gpu.plan

Plan files, fast network type:

- attributes_estimation_fast_<version>_cpu.plan
- attributes_estimation_fast_<version>_cpu-avx2.plan
- attributes_estimation_fast_<version>_gpu.plan

6.4.2 Credibility Check Estimation

Name: CredibilityCheckEstimator

Algorithm description:

This estimator estimates reliability of a person.

Implementation description:

The estimator (see ICredibilityCheckEstimator in ICredibilityCheckEstimator.h):

- Implements the *estimate()* function that accepts **warped image** in R8B8G8 format and `fsdk::CredibilityCheckEstimation` structure.
- Implements the *estimate()* function that accepts span of **warped images** in R8B8G8 format and span of `fsdk::CredibilityCheckEstimation` structures.

The **CredibilityCheckEstimation structure** contains results of the estimation:

```
struct CredibilityCheckEstimation {
    float value;                         //!< estimation in [0,1] range
        .
                                         //!< The closer the score to
                                            1,
                                         //!< the more likely that
                                            person is reliable.

    CredibilityStatus credibilityStatus;  //!< estimation result
                                         //!< (@see CredibilityStatus
                                            enum).
};
```

Enumeration of possible credibility statuses:

```
enum class CredibilityStatus : uint8_t {
    Reliable = 1,                         //!< person is reliable
    NonReliable = 2                       //!< person is not reliable
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `CredibilityEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 8: "Credibility check estimator recommended threshold"

Threshold	Recommended value
reliableThreshold	0.5

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 9: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-20…20]
yaw	[-20…20]
roll	[-20…20]

Table 10: "Requirements for `fsdk::SubjectiveQuality`"

Attribute	Minimum value
blur	0.61
light	0.57

Table 11: "Requirements for `fsdk::AttributeEstimationResult`"

Attribute	Minimum value
age	18

Table 12: "Requirements for `fsdk::OverlapEstimation`"

Attribute	State
overlapped	false

Table 13: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	100

> Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

Configurations:

See the "Credibility Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ICredibilityCheckEstimator

Plan files:

- credibility_check_cpu.plan
- credibility_check_cpu-avx2.plan
- credibility_check_gpu.plan

6.5 Facial Hair Estimation

Name: FacialHairEstimator

Algorithm description:

This estimator aims to detect a facial hair type on the face in the source image. It can return the next results:

- There is no hair on the face (see `FacialHair::NoHair` field in the FacialHair enum);
- There is stubble on the face (see `FacialHair::Stubble` field in the FacialHair enum);
- There is mustache on the face (see `FacialHair::Mustache` field in the FacialHair enum);
- There is beard on the face (see `FacialHair::Beard` field in the FacialHair enum).

Implementation description:

The estimator (see IFacialHairEstimator in IFacialHairEstimator.h):

- Implements the *estimate()* function that accepts **source warped image** in R8G8B8 format and FacialHairEstimation structure to return results of estimation;
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source warped images** in R8G8B8 format and `fsdk::Span` of the FacialHairEstimation structures to return results of estimation.

The **FacialHair enumeration** contains all possible results of the FacialHair estimation:

```
enum class FacialHair {
    NoHair = 0,                 //!< no hair on the face
    Stubble,                    //!< stubble on the face
    Mustache,                   //!< mustache on the face
    Beard                       //!< beard on the face
};
```

The **FacialHairEstimation structure** contains results of the estimation:

```
struct FacialHairEstimation {
    FacialHair result;          //!< estimation result (@see FacialHair
        enum)
    // scores
    float noHairScore;          //!< no hair on the face score
    float stubbleScore;         //!< stubble on the face score
    float mustacheScore;        //!< mustache on the face score
    float beardScore;           //!< beard on the face score
};
```

There are two groups of the fields:

1 The first group contains only the result enum:

```
            FacialHair result;          //!< estimation result (@see FacialHair
    enum)
```

Result enum field FacialHairEstimation contain the target results of the estimation.

2 The second group contains scores:

```
            float noHairScore;          //!< no hair on the face score
            float stubbleScore;         //!< stubble on the face score
            float mustacheScore;        //!< mustache on the face score
            float beardScore;           //!< beard on the face score
```

The scores group contains the estimation scores for each possible result of the estimation.

All scores are defined in [0,1] range. Sum of scores always equals 1.

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 14: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-40…40]
yaw	[-40…40]
roll	[-40…40]

Table 15: "Requirements for `fsdk::MedicalMaskEstimation`"

Attribute	State
result	fsdk::MedicalMask::NoMask

Table 16: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	40

> Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

API structure name:

IFacialHairEstimator

Plan files:

- face_hair_v2_cpu.plan
- face_hair_v2_cpu-avx2.plan
- face_hair_v2_gpu.plan

6.6 Natural Light Estimation

Name: NaturalLightEstimator

Algorithm description:

This estimator aims to detect a natural light on the source face image. It can return the next results:

- Light is not natural on the face image (see `LightStatus::NonNatural` field in the LightStatus enum);
- Light is natural on the face image (see `LightStatus::Natural` field in the LightStatus enum).

Implementation description:

The estimator (see INaturalLightEstimator in INaturalLightEstimator.h):

- Implements the *estimate()* function that accepts **source warped image** in R8G8B8 format and NaturalLightEstimation structure to return results of estimation;
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source warped images** in R8G8B8 format and `fsdk::Span` of the NaturalLightEstimation structures to return results of estimation.

The **LightStatus enumeration** contains all possible results of the NaturalLight estimation:

```
enum class LightStatus : uint8_t {
    NonNatural = 0,                 //!< light is not natural
    Natural = 1                     //!< light is natural
};
```

The **NaturalLightEstimation structure** contains results of the estimation:

```
struct NaturalLightEstimation {
    LightStatus status;             //!< estimation result (@see
       NaturalLight enum).
    float score;                    //!< Numerical value in range [0,
       1].
};
```

There are two groups of the fields:

1 The first group contains only the result enum:

```
    LightStatus status;             //!< estimation result (@see
       LightStatus enum).
```

Result enum field NaturalLightEstimation contain the target results of the estimation.

2) The second group contains scores:

```
            float score;                        //!< Numerical value in range [0,
    1].
```

The scores group contains the estimation scores for each possible result of the estimation.

All scores are defined in [0,1] range. Sum of scores always equals 1.

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `NaturalLightEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 17: "Natural light estimator recommended threshold"

Threshold	Recommended value
naturalLightThreshold	0.5

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 18: "Requirements for `fsdk::MedicalMaskEstimation`"

Attribute	State
result	fsdk::MedicalMask::NoMask

Table 19: "Requirements for `fsdk::SubjectiveQuality`"

Attribute	Minimum value
blur	0.5

Also `fsdk::GlassesEstimation` must not be equal to `fsdk::GlassesEstimation::SunGlasses`.

Configurations:

See the "Natural Light Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

INaturalLightEstimator

Plan files:

- natural_light_cpu.plan
- natural_light_cpu-avx2.plan
- natural_light_gpu.plan

6.7 Fish Eye Estimation

Name: FishEyeEstimator

Algorithm description:

This estimator aims to detect a fish eye effect on the source face image. It can return the next fish eye effect status results:

- There is no fish eye effect on the face image (see `FishEye::NoFishEyeEffect` field in the FishEye enum);
- There is fish eye effect on the face image (see `FishEye::FishEyeEffect` field in the FishEye enum).

Implementation description:

The estimator (see IFishEyeEstimator in IFishEyeEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format, face detection and FishEyeEstimation structure to return results of estimation;
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source images** in R8G8B8 format, `fsdk::Span` of the face detections and `fsdk::Span` of the FishEyeEstimation structures to return results of estimation.

The **FishEye enumeration** contains all possible results of the FishEye estimation:

```
enum class FishEye {
    NoFishEyeEffect = 0,  //!< no fish eye effect
    FishEyeEffect = 1     //!< with fish eye effect
};
```

The **FishEyeEstimation structure** contains results of the estimation:

```
struct FishEyeEstimation {
    FishEye result;       //!< estimation result (@see FishEye enum)
    float score;          //!< fish eye effect score
};
```

There are two groups of the fields:

1⍰ The first group contains only the result enum:

```
    FishEye result;       //!< estimation result (@see FishEye enum)
```

Result enum field FishEyeEstimation contain the target results of the estimation.

2 The second group contains scores:

```
float score;            //!< fish eye effect score
```

The scores group contains the estimation score.

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `FishEyeEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 20: "Fish Eye estimator recommended threshold"

Threshold	Recommended value
fishEyeThreshold	0.5

Recommended scenarios of algorithm usage:

Data domain: Cooperative mode only. It is means:

- High image quality;
- Frontal face looking directly at the camera.

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 21: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	**Acceptable angle range(degrees)**
pitch	[-8…8]
yaw	[-8…8]
roll	[-8…8]

Table 22: "Requirements for `fsdk::Detection`"

Attribute	**Minimum value**
detection size	80

> Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

Configurations:

See the “Fish Eye Estimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

IFishEyeEstimator

Plan files:

- fisheye_v2_cpu.plan
- fisheye_v2_cpu-avx2.plan
- fisheye_v2_gpu.plan

6.8 Eyebrows Estimation

Name: EyeBrowEstimator

Algorithm description:

This estimator is trained to estimate eyebrow expressions. The EyeBrowEstimator returning four scores for each possible eyebrow expression. Which are - `neutral`, `raised`, `squinting`, `frowning`. Possible scores are in the range [0, 1].

If score closer to `1`, it means that detected expression on image is more likely to real expression and closer to 0 otherwise.

Along with the output score value estimator also returns an enum value (EyeBrowState). The index of the maximum score determines the EyeBrow state.

Implementation description:

The estimator (see IEyeBrowEstimator in IEyeBrowEstimator.h):

- Implements the *estimate()* function accepts **warped source image**. Warped image is received from the warper (see `IWarper::warp()`); Output estimation is a structure `fsdk::EyeBrowEstimation` .
- Implements the *estimate()* function that needs the span of **warped source images** and span of structure `fsdk::EyeBrowEstimation`. Output estimation is a span of structure `fsdk::EyeBrowEstimation`.

The **EyeBrowEstimation structure** contains results of the estimation:

```
struct EyeBrowEstimation {
        /**
         * @brief EyeBrow estimator output enum.
         * This enum contains all possible estimation results.
        **/
        enum class EyeBrowState {
            Neutral = 0,
            Raised,
            Squinting,
            Frowning
        };


        float neutralScore;         //!< 0(not neutral)..1(neutral).
        float raisedScore;          //!< 0(not raised)..1(raised).
        float squintingScore;       //!< 0(not squinting)..1(squinting).
        float frowningScore;        //!< 0(not frowning)..1(frowning).
        EyeBrowState eyeBrowState; //!< EyeBrow state
```

```
    };
```

Filtration parameters:

Table 23: "Requirements for `fsdk::EyeBrowEstimation`"

Attribute	Acceptable values
headPose.pitch	[-20…20]
headPose.yaw	[-20…20]
headPose.roll	[-20…20]

Table 24: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

> Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

API structure name:

IEyeBrowEstimator

Plan files:

- eyebrow_estimation_v2_cpu.plan
- eyebrow_estimation_v2_cpu-avx2.plan
- eyebrow_estimation_v2_gpu.plan

6.9 Portrait Style Estimation

Name: PortraitStyleEstimator

Algorithm description:

This estimator is designed to estimate the position of a person's shoulders in the original image. It can return the following results:

- The shoulders are not parallel to the camera (see the `PortraitStyleStatus::NonPortrait` field in the PortraitStyleStatus enum);
- Shoulders are parallel to the camera (see the `PortraitStyleStatus::Portrait` field in the PortraitStyleStatus enum);
- Shoulders are hidden (see the `PortraitStyleStatus::HiddenShoulders` field in the PortraitStyleStatus enum);

Implementation description:

The Estimator (see IPortraitStyleEstimator in IPortraitStyleEstimator.h):

- Implements *estimate()* function that accepts R8G8B8 **source image**, detection and PortraitStyleEstimation structure to return estimation results;
- Implements an *estimate()* function that accepts `fsdk::Span` of R8G8B8 **source images**, `fsdk::Span` of detections, and `fsdk::Span` of PortraitStyleEstimation structures to return estimation results.

The **PortraitStyleStatus enumeration** contains all possible results of the PortraitStyle estimation:

```
enum class PortraitStyleStatus : uint8_t {
    NonPortrait = 0,        //!< NonPortrait
    Portrait = 1,           //!< Portrait
    HiddenShoulders = 2     //!< HiddenShoulders
};
```

The **PortraitStyleEstimation structure** contains results of the estimation:

```
struct PortraitStyleEstimation {
    PortraitStyleStatus status; //!< estimation result (@see
        PortraitStyleStatus enum).
    float nonPortraitScore;             //!< numerical value in range
        [0, 1]
    float portraitScore;                //!< numerical value in range
        [0, 1]
    float hiddenShouldersScore;         //!< numerical value in range
        [0, 1]
```

```
    };
```

There are two groups of the fields:

1⌷ The first group contains the enum:

```
        PortraitStyleStatus status; //!< estimation result (@see
            PortraitStyleStatus enum).
```

Result enum field PortraitStyleStatus contain the target results of the estimation.

2⌷ The second group contains score:

```
        float nonPortraitScore;             //!< numerical value in range
            [0, 1]
        float portraitScore;                //!< numerical value in range
            [0, 1]
        float hiddenShouldersScore;         //!< numerical value in range
            [0, 1]
```

The scores are defined in [0,1] range.

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `PortraitStyleEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 25: "Portrait Style estimator recommended threshold"

Threshold	Recommended value
notPortraitStyleThreshold	0.2
portraitStyleThreshold	0.35
hiddenShouldersThreshold	0.2

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Type of preferable detector is FaceDetV3.

Table 26: "Requirements for `Detector`"

Attribute	Min face size
result	40

Table 27: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Maximum value
yaw	20.0
pitch	20.0
roll	20.0

Configurations:

See the "Portrait Style Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IPortraitStyleEstimator

Plan files:

- portrait_style_v3_cpu.plan
- portrait_style_v3_cpu-avx2.plan
- portrait_style_v3_gpu.plan

6.10 DynamicRange Estimation

Name: DynamicRangeEstimator

Algorithm description:

This estimator is designed to estimate dynamic range of an original image with person's face.

Implementation description:

The Estimator (see IDynamicRangeEstimator in IDynamicRangeEstimator.h):

- Implements *estimate()* function that accepts R8G8B8 **source image**, detection and DynamicRangeEstimation structure to return estimation results;
- Implements an *estimate()* function that accepts `fsdk::Span` of R8G8B8 **source images**, `fsdk::Span` of detections, and `fsdk::Span` of DynamicRangeEstimation structures to return estimation results.

The **DynamicRangeEstimation structure** contains results of the estimation:

```
struct DynamicRangeEstimation {
    float dynamicRangeScore;            //!< numerical value in range
        [0, 1]
};
```

Result estimation DynamicRangeEstimation contains the target score.

```
    float dynamicRangeScore;            //!< numerical value in range
        [0, 1]
```

The score is defined in [0,1] range.

Recommended thresholds:

Table below contains recommended user's threshold.

Table 28: "Dynamic Range estimator recommended threshold"

Threshold	Recommended value
threshold	0.5

API structure name:

IDynamicRangeEstimator

Plan files:

DynamicRangeEstimator does not use any additional models (plans, files and etc.), this is an ISO-based algorithm that is currently only implemented on CPU devices.

6.11 Headwear Estimation

Name: HeadWearEstimator

Algorithm description:

This estimator aims to detect a headwear status and headwear type on the face in the source image. It can return the next headwear status results:

- There is headwear (see `HeadWearState::Yes` field in the HeadWearState enum);
- There is no headwear (see `HeadWearState::No` field in the HeadWearState enum);

And this headwear type results:

- There is no headwear on the head (see `HeadWearType::NoHeadWear` field in the HeadWearType enum);
- There is baseball cap on the head (see `HeadWearType::BaseballCap` field in the HeadWearType enum);
- There is beanie on the head (see `HeadWearType::Beanie` field in the HeadWearType enum);
- There is peaked cap on the head (see `HeadWearType::PeakedCap` field in the HeadWearType enum);
- There is shawl on the head (see `HeadWearType::Shawl` field in the HeadWearType enum);
- There is hat with ear flaps on the head (see `HeadWearType::HatWithEarFlaps` field in the HeadWearType enum);
- There is helmet on the head (see `HeadWearType::Helmet` field in the HeadWearType enum);
- There is hood on the head (see `HeadWearType::Hood` field in the HeadWearType enum);
- There is hat on the head (see `HeadWearType::Hat` field in the HeadWearType enum);
- There is something other on the head (see `HeadWearType::Other` field in the HeadWearType enum);

Implementation description:

The estimator (see IHeadWearEstimator in IHeadWearEstimator.h):

- Implements the *estimate()* function that accepts **warped image** in R8G8B8 format and HeadWearEstimation structure to return results of estimation;
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source warped images** in R8G8B8 format and `fsdk::Span` of the HeadWearEstimation structures to return results of estimation.

The **HeadWearState enumeration** contains all possible results of the Headwear state estimation:

```
enum class HeadWearState {
    Yes = 0,            //< there is headwear
    No,                 //< there is no headwear
    Count
```

```
};
```

The **HeadWearType enumeration** contains all possible results of the Headwear type estimation:

```
enum class HeadWearType : uint8_t {
    NoHeadWear = 0,     //< there is no headwear on the head
    BaseballCap,        //< there is baseball cap on the head
    Beanie,             //< there is beanie on the head
    PeakedCap,          //< there is peaked cap on the head
    Shawl,              //< there is shawl on the head
    HatWithEarFlaps,    //< there is hat with ear flaps on the head
    Helmet,             //< there is helmet on the head
    Hood,               //< there is hood on the head
    Hat,                //< there is hat on the head
    Other,              //< something other is on the head
    Count
};
```

The **HeadWearStateEstimation structure** contains results of the Headwear state estimation:

```
struct HeadWearStateEstimation {
    HeadWearState result; //!< estimation result (@see HeadWearState
        enum)
    float scores[static_cast<int>(HeadWearState::Count)]; //!<
        estimation scores

    /**
     * @brief Returns score of required headwear state.
     * @param [in] state headwear state.
     * @see HeadWearState for more info.
     * */
    inline float getScore(HeadWearState state) const;
};
```

There are two groups of the fields:

1⌧ The first group contains only the result enum:

```
    HeadWearState result; //!< estimation result (@see HeadWearState
        enum)
```

2⌧ The second group contains scores:

```
        float scores[static_cast<int>(HeadWearState::Count)]; //!<
            estimation scores
```

The **HeadWearTypeEstimation structure** contains results of the Headwear type estimation:

```
    struct HeadWearTypeEstimation {
        HeadWearType result; //!< estimation result (@see HeadWearType enum)
        float scores[static_cast<int>(HeadWearType::Count)]; //!< estimation
             scores

        /**
         * @brief Returns score of required headwear type.
         * @param [in] type headwear type.
         * @see HeadWearType for more info.
         * */
        inline float getScore(HeadWearType type) const;
    };
```

There are two groups of the fields:

1⌧ The first group contains only the result enum:

```
        HeadWearType result; //!< estimation result (@see HeadWearType enum)
```

2⌧ The second group contains scores:

```
        float scores[static_cast<int>(HeadWearType::Count)]; //!< estimation
             scores
```

The **HeadWearEstimation structure** contains results of both Headwear state and type estimations:

```
    struct HeadWearEstimation {
        HeadWearStateEstimation state;  //!< headwear state estimation
                                        //!< (@see HeadWearStateEstimation)
        HeadWearTypeEstimation type;    //!< headwear type estimation
                                        //!< (@see HeadWearTypeEstimation)
    };
```

The scores group contains the estimation scores for each possible result of the estimation. All scores are defined in [0,1] range. Sum of scores always equals 1.

Filtration parameters:

Table 29: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

Note. Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

API structure name:

IHeadWearEstimator

Plan files:

- head_wear_v2_cpu.plan
- head_wear_v2_cpu-avx2.plan
- head_wear_v2_gpu.plan

6.12 Background Estimation

Name: BackgroundEstimator

Algorithm description:

This estimator is designed to estimate the background in the original image. It can return the following results:

- The background is non-solid (see the `BackgroundStatus::NonSolid` field in the BackgroundStatus enum);
- The background is solid (see the `BackgroundStatus::Solid` field in the BackgroundStatus enum);

Implementation description:

The estimator (see IBackgroundEstimator in IBackgroundEstimator.h):

- Implements an *estimate()* function that accepts R8G8B8 **source image**, detection and BackgroundEstimation structure to return estimation results;
- Implements an *estimate()* function that accepts `fsdk::Span` of R8G8B8 **source images**, `fsdk::Span` of detections, and `fsdk::Span` of BackgroundEstimation structures to return estimation results.

The **BackgroundStatus enumeration** contains all possible results of the Background estimation:

```
enum class BackgroundStatus : uint8_t {
    NonSolid = 0,      //!< NonSolid
    Solid = 1          //!< Solid
};
```

The **BackgroundEstimation structure** contains results of the estimation:

```
struct BackgroundEstimation {
    BackgroundStatus status;    //!< estimation result (@see
        BackgroundStatus enum).
    float backgroundScore;      //!< numerical value in range [0, 1],
        where 1 - is uniform background, 0 - is non uniform.
    float backgroundColorScore; //!< numerical value in range [0, 1],
        where 1 - is light background, 0 - is too dark.
};
```

There are two groups of the fields:

1⃞ The first group contains the enum:

```
            BackgroundStatus status;    //!< estimation result (@see
               BackgroundStatus enum).
```

Result enum field BackgroundStatus contain the target results of the estimation.

2⊠ The second group contains scores:

```
            float backgroundScore;      //!< numerical value in range [0, 1],
               where 1 - is solid background, 0 - is non solid.
            float backgroundColorScore; //!< numerical value in range [0, 1],
               where 1 - is light background, 0 - is too dark.
```

The scores are defined in the [0,1] range. If two scores are above the threshold, then the background is solid, otherwise the background is not solid.

Recommended thresholds:

The table below contains thresholds specified in `BackgroundEstimator::Settings` section of the FaceEngine configuration file (*faceengine.conf*). By default, these threshold values are set to optimal.

Table 30: "Background estimator recommended thresholds"

Threshold	Recommended value
backgroundThreshold	0.5
backgroundColorThreshold	0.3

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements: The face in a frame should be large in relation to frame sizes. The face should occupy about half of the frame area.

```
    max(frameWidth, frameHeight) / max(faceWidth, faceHeight) <= 2.0
```

The type of preferable detector is FaceDetV3.

Table 31: "Requirements for `Detector`"

Attribute	**Min face size**
result	40

Table 32: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Maximum value
yaw	20.0
pitch	20.0
roll	20.0

Configurations:

See the "Background Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IBackgroundEstimator

Plan files:

- background_v2_cpu.plan
- background_v2_cpu-avx2.plan
- background_v2_gpu.plan

6.13 Grayscale, color or infrared Estimation

Name: BlackWhiteEstimator

Algorithm description:

BlackWhite estimator has two interfaces.

The "By full frame" interface detects if an input image is grayscale or color. It is indifferent to image content and dimensions; you can pass both face crops (including warped images) and full frames.

The "By warped frame" interface can be used only with warped images (see chapter "Image warping" for details). Checks if an image is color, grayscale or infrared.

Implementation description:

The "By full frame" interface of estimator (see ImageColorEstimation in IBlackWhiteEstimator.h):

- Implements *estimate()* function that accepts **source image** and outputs a boolean, indicating if the image is grayscale (true) or not (false).

The "By warped frame" interface of estimator (see IBlackWhiteEstimator in IBlackWhiteEstimator.h):

- Implements the *estimate()* function that accepts **warped source image**.
- Outputs ImageColorEstimation structures.

```
struct ImageColorEstimation {

    float colorScore;       //!< 0(grayscale)..1(color);
    float infraredScore;    //!< 0(infrared)..1(not infrared);

    /**
     * @brief Enumeration of possible image color types.
     * */
    enum class ImageColorType : uint8_t {
        Color = 0,      //!< image is color.
        Grayscale,      //!< Image is grayscale.
        Infrared,       //!< Image is infrared.
    };

    ImageColorType colorType;
};
```

`ImageColorEstimation::ImageColorType` presents color image type as enum with possible values: Color, Grayscale, Infrared.

```
- For color image score `colorScore` will be close to 1.0 and the second one
    `infraredScore` - to 0.0;
- for infrared image score `colorScore` will be close to 0.0 and the second
   one `infraredScore` - to 1.0;
- for grayscale images both of scores will be near 0.0.
```

> Both interfaces use different principles of color type estimation.

> BlackWhite estimator is trained to work with real warped photo of faces. We do not guarantee correctness when the people in the photo are fake (not real, such as the photo in the photo).

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `BlackWhiteEstimator::Settings` section. By default, these threshold values are set to optimal.

Table 33: "Black and white estimator recommended thresholds"

Threshold	Recommended value
colorThreshold	0.5
irThreshold	0.5

Configurations:

See the "BlackWhite Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IBlackWhiteEstimator

Plan files:

- black_white_and_ir_v1_cpu.plan
- black_white_and_ir_v1_cpu-avx2.plan
- black_white_and_ir_v1_gpu.plan

6.14 Face features extraction functionality

6.14.1 Eyes Estimation

Name: EyeEstimator

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details).

For this type of estimator can be defined sensor type.

This estimator aims to determine:

- Eye state: Open, Closed, Occluded;
- Precise eye iris location as an array of landmarks;
- Precise eyelid location as an array of landmarks.

You can only pass warped image with detected face to the estimator interface. Better image quality leads to better results.

Eye state classifier supports three categories: "Open", "Closed", "Occluded". Poor quality images or ones that depict obscured eyes (think eyewear, hair, gestures) fall into the "Occluded" category. It is always a good idea to check eye state before using the segmentation result.

The precise location allows iris and eyelid segmentation. The estimator is capable of outputting iris and eyelid shapes as an array of points together forming an ellipsis. You should only use segmentation results if the state of that eye is "Open".

Implementation description:

The estimator:

- Implements the *estimate()* function that accepts **warped source image** and warped landmarks, either of type Landmarks5 or Landmarks68. The warped image and landmarks are received from the warper (see `IWarper::warp()`);
- Classifies eyes state and detects its iris and eyelid landmarks;
- Outputs EyesEstimation structures.

> Orientation terms "left" and "right" refer to the way you see the *image* as it is shown on the screen. It means that left eye is not necessarily left from the person's point of view, but is on the left side of the screen. Consequently, right eye is the one on the right side of the screen. More formally, the label "left" refers to subject's left eye (and similarly for the right eye), such that xright < xleft.

`EyesEstimation::EyeAttributes` presents eye state as enum EyeState with possible values: Open, Closed, Occluded.

Iris landmarks are presented with a template structure Landmarks that is specialized for 32 points.

Eyelid landmarks are presented with a template structure Landmarks that is specialized for 6 points.

The **EyesEstimation structure** contains results of the estimation:

```
struct EyesEstimation {
    /**
     * @brief Eyes attribute structure.
     * */
    struct EyeAttributes {
        /**
         * @brief Enumeration of possible eye states.
         * */
        enum class State : uint8_t {
            Closed,     //!< Eye is closed.
            Open,       //!< Eye is open.
            Occluded    //!< Eye is blocked by something not transparent
                , or landmark passed to estimator doesn't point to an eye
                .
        };

        static constexpr uint64_t irisLandmarksCount = 32; //!< Iris
            landmarks amount.
        static constexpr uint64_t eyelidLandmarksCount = 6; //!< Eyelid
            landmarks amount.

        /// @brief alias for @see Landmarks template structure with
            irisLandmarksCount as param.
        using IrisLandmarks = Landmarks<irisLandmarksCount>;

        /// @brief alias for @see Landmarks template structure with
            eyelidLandmarksCount as param
        using EyelidLandmarks = Landmarks<eyelidLandmarksCount>;

        State state; //!< State of an eye.

        IrisLandmarks iris; //!< Iris landmarks.
        EyelidLandmarks eyelid; //!< Eyelid landmarks
    };

    EyeAttributes leftEye;  //!< Left eye attributes
    EyeAttributes rightEye; //!< Right eye attributes
};
```

API structure name:

IEyeEstimator

Plan files:

- eyes_estimation_flwr8_cpu.plan
- eyes_estimation_ir_cpu.plan
- eyes_estimation_flwr8_cpu-avx2.plan
- eyes_estimation_ir_cpu-avx2.plan
- eyes_estimation_ir_gpu.plan
- eyes_estimation_flwr8_gpu.plan
- eye_status_estimation_cpu.plan
- eye_status_estimation_cpu-avx2.plan
- eye_status_estimation_gpu.plan

6.14.2 Red Eyes Estimation

Name: RedEyeEstimator

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details) and warped landmarks.

Red Eye estimator evaluates whether a person's eyes are red in a photo or not.

You can pass only warped images with detected faces to the estimator interface. Better image quality leads to better results.

Implementation description:

The estimator (see IRedEyeEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts **warped source image** in R8G8B8 format and warped Landmarks5. The warped image and landmarks are received from the warper (see `IWarper::warp()`);.
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source warped images** in R8G8B8 format and `fsdk::Span` of warped Landmarks.
- Outputs RedEyeEstimation structure.

RedEyeEstimation structure consists of attributes for each eye. Eye attributes consists of a score of and status. Scores is normalized float value in a range of [0..1] where 1 is red eye and 0 is not.

The **RedEyeEstimation structure** contains results of the estimation:

```
struct RedEyeEstimation {
    /**
     * @brief Eyes attribute structure.
     * */
    struct RedEyeAttributes {
        RedEyeStatus status;    //!< Status of an eye.
        float score;            //!< Score, numerical value in range
            [0,1].
    };

    RedEyeAttributes leftEye;  //!< Left eye attributes
    RedEyeAttributes rightEye; //!< Right eye attributes
};
```

There are two groups of the fields in RedEyeAttributes:

1⊠ The first field is a status:

```
        RedEyeStatus status;    //!< Status of an eye.
```

2⊠ The second field is a score, which defined in [0,1] range:

```
        float score;        //!< Score, numerical value in range [0, 1].
```

Enumeration of possible red eye statuses.

```
    enum class RedEyeStatus : uint8_t {
        NonRed,     //!< Eye is not red.
        Red,        //!< Eye is red.
    };
```

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `RedEyeEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 34: "Red eye estimator recommended threshold"

Threshold	Recommended value
redEyeThreshold	0.5

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 35: "Requirements for `fsdk::NaturalLight`"

Attribute	Minimum value
score	0.5

Table 36: "Requirements for `fsdk::SubjectiveQuality`"

Attribute	Minimum value
blur	0.61

Attribute	Minimum value
light	0.57
darkness	0.5
illumination	0.1
specularity	0.1

Also `fsdk::GlassesEstimation` must not be equal to `fsdk::GlassesEstimation::SunGlasses`.

Configurations:

See the "RedEyeEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IRedEyeEstimator

Plan files:

- red_eye_v1_cpu.plan
- red_eye_v1_cpu-avx2.plan
- red_eye_v1_gpu.plan

6.14.3 Gaze Estimation

Name: GazeEstimator

Algorithm description:

This estimator is designed to determine gaze direction relatively to head pose estimation. Since 3D head translation is hard to determine reliably without camera-specific calibration, only 3D rotation component is estimated.

For this type of estimator can be defined sensor type.

Estimation characteristics:

- Units (degrees);
- Notation (Euler angles);
- Accuracy (see table below).

> Roll angle is not estimated, prediction accuracy decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table below.

Implementation description:

The **GazeEstimation structure** contains results of the estimation. Each angle is measured in degrees and in [-180, 180] range:

```
struct GazeEstimation {
    float yaw;      //!< Eye yaw angle.
    float pitch;    //!< Eye pitch angle.
};
```

Metrics:

Table below contains gaze prediction accuracy values.

Table 37: "Gaze prediction accuracy"

	Range	-25°...+25°	-25° ... -45 ° or 25 ° ... +45°
Average prediction error (per axis)	Yaw	±2.7°	±4.6°
Average prediction error (per axis)	Pitch	±3.0°	±4.8°

Zero position corresponds to a gaze direction orthogonally to face plane, with the axis of symmetry parallel to the vertical camera axis.

API structure name:

IGazeEstimator

Plan files:

- gaze_v2_cpu.plan
- gaze_v2_cpu-avx2.plan
- gaze_v2_gpu.plan
- gaze_ir_v2_cpu.plan
- gaze_ir_v2_cpu-avx2.plan
- gaze_ir_v2_gpu.plan

6.15 Head Pose Estimation

This estimator is designed to determine a camera-space head pose. Since the 3D head translation is hard to reliably determine without a camera-specific calibration, only the 3D rotation component is estimated.

There are two head pose estimation methods available:

- Estimate by 68 face-aligned landmarks. You can get it from the Detector facility, see Chapter "Face detection facility" for details.
- Estimate by the original input image in the RGB format.

An estimation by the image is more precise. If you have already extracted 68 landmarks for another facilities, you can save time and use the fast estimator from 68 landmarks.

By default, all methods are available to use in the faceengine.conf configuration file in section "HeadPoseEstimator". You can disable these methods to decrease RAM usage and initialization time.

Estimation characteristics:

- Units (degrees)
- Notation (Euler angles)
- Precision (see table 38)

> **Note:** Prediction precision decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table 38.

Table 38: "Head pose prediction precision"

	Range	-45°...+45°	< -45° or > +45°
Average prediction error (per axis)	Yaw	±2.7°	±4.6°
Average prediction error (per axis)	Pitch	±3.0°	±4.8°
Average prediction error (per axis)	Roll	±3.0°	±4.6°

Zero position corresponds to a face placed orthogonally to the camera direction, with the axis of symmetry parallel to the vertical camera axis. See figure 21 for a reference.

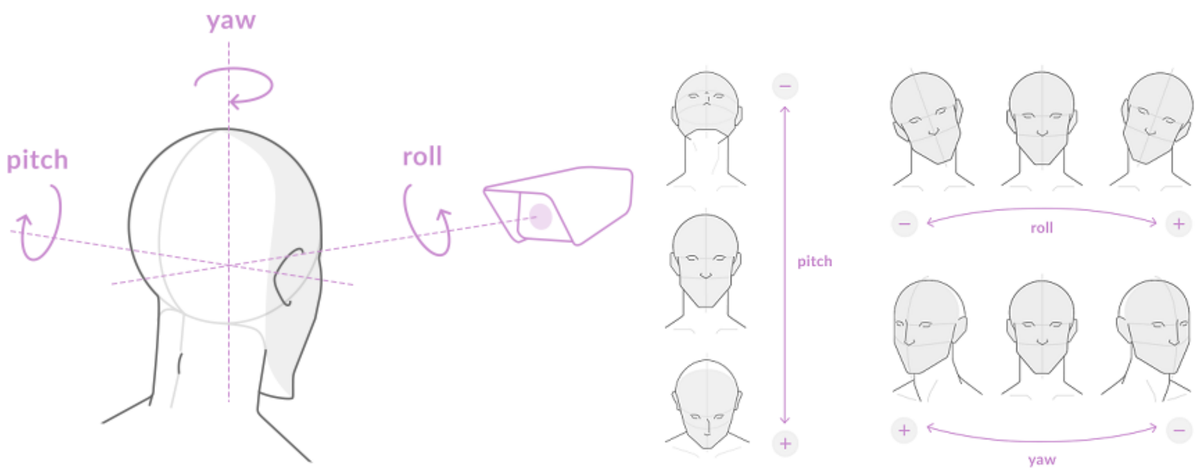


Figure 21: Head pose illustration

> **Note:** In order to work, this estimator requires precise 68-point face alignment results, so familiarize with section "Face alignment" in the "Face detection facility" chapter, as well.

6.16 Approximate Garbage Score Estimation (AGS)

This estimator aims to determine the source image score for further descriptor extraction and matching. The higher the score, the better matching result is received for the image.

When you have several images of a person, it is better to save the image with the highest AGS score.

> Contact VisionLabs for the recommended threshold value for this parameter.

The estimator (see `IAGSEstimator` in `IEstimator.h`):

- Implements the `estimate()` function that accepts the source image in the R8G8B8 format and the `fsdk::Detection` structure of corresponding source image. For details, see section "Detection structure" in chapter "Face detection facility".
- Estimates garbage score of the input image.
- Outputs a garbage score value.

6.16.1 Glasses Estimation

Name: GlassesEstimator

Algorithm description:

Glasses estimator is designed to determine whether a person is currently wearing any glasses or not. There are 3 types of states the estimator is currently able to estimate:

- `NoGlasses` - Determines whether a person is wearing any glasses at all.
- `EyeGlasses` - Determines whether a person is wearing eyeglasses.
- `SunGlasses` - Determines whether a person is wearing sunglasses.

Note: The source input image must be warped for the estimator to work properly (see chapter "Image warping" for details). Estimation quality depends on threshold values located in the faceengine.conf configuration file.

Implementation description:

Enumeration of possible glasses estimation statuses:

```
enum class GlassesEstimation: uint8_t{
    NoGlasses,      //!< Person is not wearing glasses
    EyeGlasses,     //!< Person is wearing eyeglasses
    SunGlasses,     //!< Person is wearing sunglasses
    EstimationError //!< failed to estimate
};
```

Recommended thresholds:

The table below contains thresholds specified in `GlassesEstimator::Settings` section of the FaceEngine configuration file (*faceengine.conf*). By default, these threshold values are set to optimal.

Table 39: "Glasses estimator recommended thresholds"

Threshold	Recommended value
noGlassesThreshold	1
eyeGlassesThreshold	1
sunGlassesThreshold	1

Configurations:

See the "GlassesEstimator settings" section in the "ConfigurationGuide.pdf" document.

Metrics:

The table below contains true positive rates corresponding to the selected false positive rates.

Table 40: "Glasses estimator TPR/FPR rates"

State	TPR	FPR
NoGlasses	0.997	0.00234
EyeGlasses	0.9768	0.000783
SunGlasses	0.9712	0.000383

API structure name:

IGlassesEstimator

Plan files:

- glasses_estimation_v2_cpu.plan
- glasses_estimation_v2_cpu-avx2.plan
- glasses_estimation_v2_gpu.plan

6.16.2 Overlap Estimation

Name: OverlapEstimator

Algorithm description:

This estimator tells whether the face is overlapped by any object. It returns a structure with value of overlapping and Boolean answer. It returns a structure with 2 fields. One is the value of overlapping in the range [0..1] where 0 is not overlapped and 1.0 is overlapped, the second is a Boolean answer. A Boolean answer depends on the threshold listed below. If the value is greater than the threshold, the answer returns true, else false.

Implementation description:

The estimator (see IOverlapEstimator in IOverlapEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format and `fsdk::Detection` structure of corresponding source image (see section "Detection structure");
- Estimates whether the face is overlapped by any object on input image;
- Outputs structure with value of overlapping and Boolean answer.

The **OverlapEstimation structure** contains results of the estimation:

```
struct OverlapEstimation {
    float overlapValue; //!< Numerical value of face overlapping in
        range [0, 1].
    bool overlapped;    //!< Overlapped face (true) or not (false).
};
```

Recommended thresholds:

Table below contains threshold from faceengine configuration file (faceengine.conf) in `OverlapEstimator::Settings` section. By default, this threshold value is set to optimal.

Table 41: "Overlap estimator recommended threshold"

Threshold	Recommended value
overlapThreshold	0.01

Configurations:

See the "OverlapEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IOverlapEstimator

Plan files:

- overlap_estimation_v1_cpu.plan
- overlap_estimation_v1_cpu-avx2.plan
- overlap_estimation_v1_gpu.plan

6.17 Emotion estimation functionality

6.17.1 Emotions Estimation

Name: EmotionsEstimator

Algorithm description:

> The estimator is trained to work with warped images (see chapter "Image warping" for details).

This estimator aims to determine whether a face depicted on an image expresses the following emotions:

- Anger
- Disgust
- Fear
- Happiness
- Surprise
- Sadness
- Neutrality

You can pass only warped images with detected faces to the estimator interface. Better image quality leads to better results.

Implementation description:

The estimator (see IEmotionsEstimator in IEmotionsEstimator.h):

- Implements the *estimate()* function that accepts **warped source image**. Warped image is received from the warper (see `IWarper::warp()`);
- Estimates emotions expressed by the person on a given image;
- Outputs EmotionsEstimation structure with aforementioned data.

EmotionsEstimation presents emotions as normalized float values in the range of [0..1] where 0 is lack of a specific emotion and 1 is the maximum intensity of an emotion.

The **EmotionsEstimation structure** contains results of the estimation:

```
struct EmotionsEstimation {
    float anger;     //!< 0(not angry)..1(angry);
    float disgust;   //!< 0(not disgusted)..1(disgusted);
    float fear;      //!< 0(no fear)..1(fear);
    float happiness;//!< 0(not happy)..1(happy);
    float sadness;   //!< 0(not sad)..1(sad);
    float surprise; //!< 0(not surprised)..1(surprised);
    float neutral;   //!< 0(not neutral)..1(neutral).

    enum Emotions {
        Anger = 0,
        Disgust,
        Fear,
        Happiness,
        Sadness,
        Surprise,
        Neutral,
        Count
    };

    /**
     * @brief Returns emotion with greatest score
     * */
    inline Emotions getPredominantEmotion() const;

    /**
     * @brief Returns score of required emotion
     * @param [in] emotion emotion
     * @see Emotions for details.
     * */
    inline float getEmotionScore(Emotions emotion) const;
};
```

API structure name:

IEmotionsEstimator

Plan files:

- emotion_recognition_v2_cpu.plan
- emotion_recognition_v2_cpu-avx2.plan
- emotion_recognition_v2_gpu.plan

6.18 Mouth Estimation Functionality

Name: MouthEstimator

Algorithm description:

This estimator is designed to predict person's mouth state.

Implementation description:

Mouth Estimation

It returns the following bool flags:

```
    bool isOpened;   //!< Mouth is opened flag
    bool isSmiling;  //!< Person is smiling flag
    bool isOccluded; //!< Mouth is occluded flag
```

Each of these flags indicate specific mouth state that was predicted.

The combined mouth state is assumed if multiple flags are set to true. For example there are many cases where person is smiling and its mouth is wide open.

Mouth estimator provides score probabilities for mouth states in case user need more detailed information:

```
    float opened;    //!< mouth opened score
    float smile;     //!< person is smiling score
    float occluded;  //!< mouth is occluded score
```

Mouth Estimation Extended

This estimation is extended version of regular Mouth Estimation (see above). In addition, It returns the following fields:

```
    SmileTypeScores smileTypeScores; //!< Smile types scores
    SmileType smileType; //!< Contains smile type if person "isSmiling"
```

If flag `isSmiling` is true, you can get more detailed information of smile using `smileType` variable. `smileType` can hold following states:

```
    enum class SmileType {
        None,  //!< No smile
        SmileLips, //!< regular smile, without teeths exposed
        SmileOpen //!< smile with teeths exposed
    };
```

If `isSmiling` is false, the `smileType` assigned to `None`. Otherwise, the field will be assigned with `SmileLips` (person is smiling with closed mouth) or `SmileOpen` (person is smiling with open mouth, with teeth's exposed).

Extended mouth estimation provides score probabilities for smile type in case user need more detailed information:

```
struct SmileTypeScores {
    float smileLips; //!< person is smiling with lips score
    float smileOpen; //!< person is smiling with open mouth score
};
```

`smileType` variable is set based on according scores hold by `smileTypeScores` variable - set based on maximum score from `smileLips` and `smileOpen` or to `None` if person not smiling at all.

```
if (estimation.isSmiling)
    estimation.smileType = estimation.smileTypeScores.smileLips >
        estimation.smileTypeScores.smileOpen ?
         fsdk::SmileType::SmileLips : fsdk::SmileType::SmileOpen;
else
    estimation.smileType = fsdk::SmileType::None;
```

> When you use Mouth Estimation Extended, the underlying computation are exactly the same as if you use regular Mouth Estimation. The regular Mouth Estimation was retained for backward compatibility.

> These estimators are trained to work with warped images (see Chapter "Image warping" for details).

Recommended thresholds:

The table below contains thresholds specified in the `MouthEstimator::Settings` section of the FaceEngine configuration file (*faceengine.conf*). By default, these threshold values are set to optimal.

Table 42: "Mouth estimator recommended thresholds"

Threshold	Recommended value
occlusionThreshold	0.5
smileThreshold	0.5
openThreshold	0.5

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

- Requirements for `Detector`:

Attribute	Minimum value
detection size	80

> Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

- Requirements for `fsdk::MouthEstimator`:

Attribute	Acceptable values
headPose.pitch	[-20…20]
headPose.yaw	[-25…25]
headPose.roll	[-10…10]

Configurations:

See the "Mouth Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IMouthEstimator

Plan files:

- mouth_estimation_v4_arm.plan
- mouth_estimation_v4_cpu.plan
- mouth_estimation_v4_cpu-avx2.plan
- mouth_estimation_v4_gpu.plan

6.19 Face Occlusion Estimation Functionality

Name: FaceOcclusionEstimator

Algorithm description:

This estimator is designed to predict occlusions in different parts of the face, such as the forehead, eyes, nose, mouth, and lower face. It also provides an overall occlusion score.

Implementation description:

Face Occlusion Estimation

The estimator returns the following occlusion states:

```
/**
 * @brief FaceOcclusionType enum.
 * This enum contains all possible facial occlusion types.
 * */
enum class FaceOcclusionType : uint8_t {
    Forehead = 0, //!< Forehead
    LeftEye,      //!< Left eye
    RightEye,     //!< Right eye
    Nose,         //!< Nose
    Mouth,        //!< Mouth
    LowerFace,    //!< Lower part of the face (chin, mouth, etc.)
    Count         //!< Total number of occlusion types
};

/**
 * @brief FaceOcclusionState enum.
 * This enum contains all possible facial occlusion states.
 * */
enum class FaceOcclusionState : uint8_t {
    NotOccluded = 0, //!< Face is not occluded
    Occluded,        //!< Face is occluded
    Count            //!< Total number of states
};


FaceOcclusionState states[static_cast<uint8_t>(FaceOcclusionType::Count)];
   //!< Occlusion states for each face region
float typeScores[static_cast<uint8_t>(FaceOcclusionType::Count)]; //!<
   Probability scores for occlusion types
FaceOcclusionState overallOcclusionState; //!< Overall occlusion state
float overallOcclusionScore;              //!< Overall occlusion score
float hairOcclusionScore;                 //!< Hair occlusion score
```

To get the occlusion score for a specific facial zone, you can use the following method:

```
float getScore(FaceOcclusionType type) const {
    return typeScores[static_cast<uint8_t>(type)];
}
```

To get the occlusion state for a specific facial zone, use the following:

```
FaceOcclusionState getState(FaceOcclusionType type) const {
    return states[static_cast<uint8_t>(type)];
}
```

> This estimator is trained to work with warped images and Landmarks5 (see Chapter "Image warping" for details).

Recommended thresholds:

The table below contains thresholds specified in the FaceOcclusion::Settings section of the FaceEngine configuration file (faceengine.conf). These values are optimal by default.

Threshold	Recommended value
normalHairCoeff	0.15
overallOcclusionThreshold	0.14
foreheadThreshold	0.2
eyeThreshold	0.4
noseThreshold	0.4
mouthThreshold	0.15
lowerFaceThreshold	0.2

Configurations

See the "Face Occlusion Estimator settings" section in the "ConfigurationGuide.pdf" document.

Filtration parameters:

Name	Threshold
Face Size	>80px
Yaw, Pitch, Roll	±20
Blur (Subjective Quality)	>0.61

API structure name:

IFaceOcclusionEstimator

Plan files:

- face_occlusion_v1_arm.plan
- face_occlusion_v1_cpu.plan
- face_occlusion_v1_cpu-avx2.plan
- face_occlusion_v1_gpu.plan

6.20 DeepFake estimation functionality

Name: DeepFakeEstimator

Algorithm description:

This estimator is designed to predict whether the face detected in the input image is synthetic or not.

Important notes:

The current implementation is experimental and does not support backward compatibility. The API can be modified in upcoming versions.

Tests were carried out with images generated by technologies from the list below:

- Deepfacelive
- FaceSwap
- Face2Face
- NeuralTextures
- FSGAN
- StyleGAN (v1, v2)
- Roop (InsightFaceSwap)
- Deepfacelab
- SimSwap (also Dot)
- FaceFusion
- MidJourney (v5, v6)
- StableDiffusion
- Faceswapper
- PicsiAI
- SwapFace
- HeyGen
- PhotoAvatar
- Vidnoz
- BlendFace
- LivePortrait
- FakeAVCeleb
- AVLips
- Recraft
- Flux-dev
- Ideogram
- DFE-2024
- NVFAIR
- pika
- wan2.1

- hailuo
- KlingAI
- Veo3
- Veo2
- MidJourney
- LatentSync
- FLUX.Kontext
- Ideogram3.0
- Sora
- Memo
- Reve
- HiggsfieldSoul
- Mocha
- Hedra

Implementation description:

`DeepFakeEstimator` returns the following structure:

```
struct DeepFakeEstimation {
    enum class State {
        Real = 0,     //!< The person in image is real
        Fake          //!< The person in image is fake (media is synthetic)
    };

    float score;      //!< Estimation score
    State state;      //!< Liveness status
};
```

The estimation score normalized between 0.0 and 1.0, where 1.0 equals to 100% confidence that media is not synthetic (*real*), and 0.0 equals to 0% that the media is synthetic (*fake*).

Requirements for a detected face in the source image:

- Minimum face height is 150 pixels.
- Yaw angles should not exceed 30 degrees.
- Pitch angles should not exceed 20 degrees.

Recommended thresholds:

The table below contains thresholds specified in `DeepFakeEstimator::Settings` section of the FaceEngine configuration file (*faceengine.conf*). By default, these threshold values are set to optimal.

Table 47: "DeepFakeEstimator recommended settings"

Parameter	Description	Type	Default value
version	The version of DeepFakeEstimator, 10 or 9	`Value::Int1`	`10`
realThreshold	Threshold in [0..1] range.	`Value::Float1`	`0.5`

Configurations:

See the "DeepFake Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

`IDeepFakeEstimator`

API namespace:

`fsdk::experimental::IDeepFakeEstimator`

Plan files, version 9:

- deepfake_estimation_v9_model_2_cpu-avx2.plan
- deepfake_estimation_v9_model_2_gpu.plan

Plan files, version 10:

- deepfake_estimation_v10_cpu-avx2.plan
- deepfake_estimation_v10_gpu.plan

Liveness check functionality

6.20.1 LivenessFlyingFaces Estimation

Name: LivenessFlyingFacesEstimator

Algorithm description:

This estimator tells whether the person's face is real or fake (photo, printed image).

Implementation description:

The estimator (see ILivenessFlyingFacesEstimator in ILivenessFlyingFacesEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Image` with valid image in R8G8B8 format and `fsdk::Detection` of corresponding **source image** (see section "Detection structure" in chapter "Face detection facility").
- Implements the *estimate()* function that needs the span of `fsdk::Image` with valid **source images** in R8G8B8 formats and span of `fsdk::Detection` of corresponding source images (see section "Detection structure" in chapter "Face detection facility").

Those methods estimate whether different persons are real or not. Corresponding estimation output with float scores which are normalized in range [0..1], where 1 - is real person, 0 - is fake.

> The estimator is trained to work in combination with `fsdk::ILivenessRGBMEstimator`.

The **LivenessFlyingFacesEstimation structure** contains results of the estimation:

```
struct LivenessFlyingFacesEstimation {
    float score;    //!< Numerical value in range [0, 1].
    bool isReal;    //!< Is real face (true) or not (false).
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in LivenessFlyingFacesEstimator section. By default, these threshold values are set to optimal.

Table 48: "Mouth estimator recommended thresholds"

Threshold	Recommended value
realThreshold	0.5
aggregationCoeff	0.7

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 49: "Requirements for `fsdk::BestShotQualityEstimator::EstimationResult`"

Attribute	Acceptable values
headPose.pitch	[-30…30]
headPose.yaw	[-30…30]
headPose.roll	[-40…40]
ags	[0.5…1.0]

Table 50: "Requirements for `fsdk::Detection`"

Attribute	Minimum value
detection size	80

> Detection size is detection width.

```
const fsdk::Detection detection = ... // somehow get fsdk::Detection object
const int detectionSize = detection.getRect().width;
```

Configurations:

See the "LivenessFlyingFaces Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessFlyingFacesEstimator

Plan files:

- flying_faces_liveness_v4_cpu.plan
- flying_faces_liveness_v4_cpu-avx2.plan
- flying_faces_liveness_v4_gpu.plan

6.20.2 LivenessRGBM Estimation

Name: LivenessRGBMEstimator

Algorithm description:

This estimator tells whether the person's face is real or fake (photo, printed image).

Implementation description:

The estimator (see ILivenessRGBMEstimator in ILivenessRGBMEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Face` with valid image in R8G8B8 format, detection structure of corresponding **source image** (see section "Detection structure" in chapter "Face detection facility") and `fsdk::Image` with accumulated background. This method estimates whether a real person or not. Output estimation structure contains the float score and boolean result. The float score normalized in range [0..1], where 1 - is real person, 0 - is fake. The boolean result has value true for real person and false otherwise.
- Implements the *update()* function that needs the `fsdk::Image` with current frame, number of that image and previously accumulated background. The accumulated background will be overwritten by this call.

The **LivenessRGBMEstimation structure** contains results of the estimation:

```
struct LivenessRGBMEstimation {
    float score = 0.0f; //!< Estimation score
    bool isReal = false;//!< Where person is real or not
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `LivenessRGBMEstimator::Settings` section. By default, these threshold values are set to optimal.

Table 51: "LivenessRGBM estimator recommended thresholds"

Threshold	Recommended value
backgroundCount	100
threshold	0.8
coeff1	0.222
coeff2	0.222

Configurations:

See the “LivenessRGBM Estimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

ILivenessRGBMEstimator

Plan files:

- rgbm_liveness_cpu.plan
- rgbm_liveness_cpu-avx2.plan
- rgbm_liveness_gpu.plan

6.20.3 Depth Liveness Estimation (LivenessDepthEstimator)

Name: LivenessDepthEstimator

Algorithm description:

This estimator tells whether the person's face is real or fake (photo, printed image).

Implementation description:

The estimator (see ILivenessDepthEstimator in ILivenessDepthEstimator.h):

- Implements the *estimate()* function that accepts **source warped image** (see chapter "Image warping" for details) in R16 format and `fsdk::DepthEstimation` structure. This method estimates whether or not depth map corresponds to the real person. Corresponding estimation output with float score which is normalized in range [0..1], where 1 - is real person, 0 - is fake.

The **DepthEstimation structure** contains results of the estimation:

```
struct DepthEstimation {
    float score; //!< confidence score in [0,1] range. The closer the
        score to 1, the more likely that person is alive.
    bool isReal; //!< boolean flag that indicates whether a person is
        real.
};
```

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in `DepthEstimator::Settings` section. By default, these threshold values are set to optimal.

Table 52: "Depth estimator recommended thresholds"

Threshold	Recommended value
maxDepthThreshold	3000
minDepthThreshold	100
zeroDepthThreshold	0.66
confidenceThreshold	0.89

Filtration parameters:

The estimator is trained to work with face images that meet the following requirements:

Table 53: "Requirements for `fsdk::HeadPoseEstimation`"

Attribute	Acceptable angle range(degrees)
pitch	[-15…15]
yaw	[-15…15]
roll	[-10…10]

Table 54: "Requirements for `fsdk::Quality`"

Attribute	Minimum value
blur	0.94
light	0.90
dark	0.93

Table 55: "Requirements for `fsdk::EyesEstimation`"

Attribute	State
leftEye	Open
rightEye	Open

Also, the minimum distance between the face bounding box and the frame borders should be greater than 20 pixels.

Configurations:

See the "Depth Estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessDepthEstimator

Plan files:

- depth_estimation_v2_1_cpu.plan
- depth_estimation_v2_1_cpu-avx2.plan
- depth_estimation_v2_1_gpu.plan

6.20.4 Depth and RGB OneShotLiveness estimation

Name: LivenessDepthRGBEstimator

Algorithm description:

This estimator shows whether the person's face is real or fake (photo, printed image). You can use this estimator in payment terminals (POS) and self-service cash registers (KCO) with two cameras - Depth and RGB.

The estimation is performed on the device with an Orbbec camera. The camera can be either built in a POS or KCO device or connected to it. This allows to perform the estimation at a higher speed and makes it more secure as data is not sent to the backend. Using the algorithm with Orbbec cameras lets you work with deep data. It increases system reliability and accuracy, as 3D data lets you assess facial shapes and detect fake masks more accurately.

The estimator is trained to work with warped images. For details, see chapter "Image warping".

Supported devices

The estimator works only on the following devices:

- VLS LUNA CAMERA 3D
- VLS LUNA CAMERA 3D Embedded

Different models of Orbbec cameras have different spacing between sensors. If you need to use another Orbbec Depth+RGB camera, you can change the calibration coefficients to match the device. Please, contact VisionLabs for details.

Image requirements

This estimator works based on two images:

- RGB image from the RGB camera
- Depth image (or depth map) from the depth camera

Input images must meet the following requirements:

Parameter	Requirements
Resolution	640 × 480 pixels
Compression	No
Image cropping	No
Image rotation	No
Effects overlay	No
Number of faces in the frame	1

Parameter	Requirements
Face detection bounding box size	200 pixels
Frame edges offset	10 pixels
Head pose	-20 to +20 degrees for head pitch, yaw, and roll.
Image quality	The face in the frame should not be overexposed, underexposed, or blurred. For details, see section "Image Quality Estimation".

Implementation description:

The estimator implements the following:

- The `estimate()` function that needs the depth frame as the first `fsdk::Image` object, the RGB frame as the second `fsdk::Image` object, `fsdk::Detection` and `fsdk::Landmarks5` objects (see section "Detection structure" in chapter "Face detection facility"). The estimation output is the `fsdk::DepthRGBEstimation` srtucture.
- The `estimate()` function that needs the first span of depth frames as the `fsdk::Image` objects, the second span of RGB frames as the `fsdk::Image` objects, a span of `fsdk::Detection`, and a span of `fsdk::Landmarks5` (see section "Detection structure" in chapter "Face detection facility").
  The estimation output is a span of the `fsdk::DepthRGBEstimation` structure. The second output value is the `fsdk::DepthRGBEstimation`structure.

`DepthRGBEstimation`

The `DepthRGBEstimation` structure contains results of the estimation:

```
struct DepthRGBEstimation {
    //!< confidence score in [0,1] range.
    //!< The closer the score to 1, the more likely that person is alive.
    float score;
    //!< boolean flag that indicates whether a person is real.
    bool isReal;
};
```

The estimation score is normalized in range [0..1], where 1 - is real person, 0 - is a fake.

The value of `isReal` depends on `score` and `confidenceThreshold`. The value of the `confidenceThreshold` can be changed in configuration file *faceengine.conf* (see *ConfigurationGuide* `LivenessDepthRGBEstimator`).

API structure name:

`ILivenessDepthRGBEstimator`

See `ILivenessDepthRGBEstimator` in `ILivenessDepthRGBEstimator.h`.

Plan files:

- depth_rgb_v2_model_1_cpu.plan
- depth_rgb_v2_model_1_gpu.plan
- depth_rgb_v2_model_2_cpu.plan
- depth_rgb_v2_model_2_gpu.plan
- depth_rgb_v2_model_1_cpu-avx2.plan
- depth_rgb_v2_model_2_cpu-avx2.plan

6.20.5 Depth liveness estimation (DepthLivenessEstimator)

Name: DepthLivenessEstimator

Algorithm description:

Given a face depth warp, the estimator tells whether the face is real or fake (photo, printed image).

The estimator aims to unify different use cases of depth liveness estimation, while increasing the estimation accuracy compared to existing depth estimators.

The estimator can be used in payment terminals (POS) and self-service cash registers (KCO) with two cameras - Depth and RGB.

The estimator is trained to work with warped depth images of faces. For details, see chapter "Image warping".

The estimator can be used together with LivenessDepthRGBEstimator or as standalone. When DepthLivenessEstimator is used in conjunction with LivenessDepthRGBEstimator, the latter takes care of necessary preprocessing of RGB and depth frames, producing depth warps of faces required by DepthLivenessEstimator. When DepthLivenessEstimator is used as standalone, it is your responsibility to prepare a warped depth image of a face for estimation, including handling such issues as:

1. detecting faces on RGB frames, quality checking of RGB frames and detections
2. [possibly required] mapping between a) RGB frames used for face detection and b) depth frames
3. obtaining depth warps of faces from depth frames

Supported devices

On its own, the estimator requires just a properly prepared depth warp of a face, and doesn't constrain the list of possible devices. However, if LivenessDepthRGBEstimator is involved, it has its own constraints.

Image requirements

The estimator works based on depth warps of faces. The warps must be 250x250 pixels, in the `fsdk::Format::R16` format. If you prepare depth warps yourself, there are some basic quality requirements for RGB frames:

Parameter	Requirements
Resolution	640 × 480 pixels
Compression	No
Image cropping	No
Image rotation	No
Effects overlay	No
Number of faces in the frame	1

Parameter	Requirements
Face detection bounding box size	200 pixels
Frame edges offset	10 pixels
Head pose	-15 to +15 degrees for head pitch, yaw, and roll.
Image quality	The face in the frame should not be overexposed, underexposed, or blurred. For details, see section "Image Quality Estimation".

Implementation description:

The estimator (see `IDepthLivenessEstimator.h`) implements the following:

- The `estimate()` function that needs the depth warp as the first `fsdk::Image` object. The estimation output is the returned `fsdk::DepthLivenessEstimation` structure.
- The `estimate()` function that needs a span of depth warps (`fsdk::Image` objects) as the first parameter, and a span of `fsdk::DepthLivenessEstimation` as the second parameter. The estimation output is saved in the second parameter.

`DepthLivenessEstimation`

The `DepthLivenessEstimation` structure contains results of the estimation:

```
struct DepthLivenessEstimation {
    //!< confidence score in [0,1] range.
    //!< The closer the score to 1, the more likely that person is alive.
    float score;
    //!< boolean flag that indicates whether a person is real.
    bool isReal;
};
```

The estimation score is normalized in the range [0..1], where 1 - is real person, 0 - is a fake.

The value of `isReal` depends on `score` and `confidenceThreshold`. The value of the `confidenceThreshold` can be changed in configuration file *faceengine.conf* (see *ConfigurationGuide* `DepthLivenessEstimator`
).

API structure name:

`IDepthLivenessEstimator`

See `IDepthLivenessEstimator` in `IDepthLivenessEstimator.h`.

Examples:

- C++ example: example_depth_liveness
- Python example: example_depth_liveness.py

Plan files:

- depth_liveness_v2_cpu.plan
- depth_liveness_v2_cpu-avx2.plan
- depth_liveness_v2_gpu.plan

6.20.6 LivenessOneShotRGB Estimation

Name: LivenessOneShotRGBEstimator

Algorithm description:

This estimator shows whether the person's face is real or fake by the following types of attacks:

- Printed Photo Attack. One or several photos of another person are used.
- Video Replay Attack. A video of another person is used.
- Printed Mask Attack. An imposter cuts out a face from a photo and covers his face with it.
- 3D Mask Attack. An imposer puts on a 3D mask depicting the face of another person.

The requirements for the processed image and the face in the image are listed below.

Parameters	Requirements
Minimum resolution for mobile devices	640x480 pixels
Maximum resolution for mobile devices	1080x1920 pixels
Minimum resolution for webcams	640x480 pixels
Maximum resolution for webcams	1920x1080 pixels
Compression	No
Image warping	No
Image cropping	No
Effects overlay	No
Mask	No
Number of faces in the frame	1
Face detection bounding box width	More than 200 pixels
Frame edges offset	More than 10 pixels
Head pose	-20 to +20 degrees for head pitch, yaw, and roll
Image quality	The face in the frame should not be overexposed, underexposed, or blurred.

See image quality thresholds in the "Image Quality Estimation" section.

Implementation description:

The estimator (see ILivenessOneShotRGBEstimator in ILivenessOneShotRGBEstimator.h):

- Implements the *estimate()* function that needs `fsdk::Image`, `fsdk::Detection` and `fsdk::Landmarks5` objects (see section "Detection structure" in chapter "Face detection facility"). Output estimation is a structure `fsdk::LivenessOneShotRGBEstimation`.
- Implements the *estimate()* function that needs the span of `fsdk::Image`, span of `fsdk::Detection` and span of `fsdk::Landmarks5` (see section "Detection structure" in chapter "Face detection facility").
  The first output estimation is a span of structure `fsdk::LivenessOneShotRGBEstimation`. The second output value (structure `fsdk::LivenessOneShotRGBEstimation`) is the result of aggregation based on span of estimations announced above. Pay attention the second output value (aggregation) is optional, i.e. `default argument`, which is `nullptr`.

The **LivenessOneShotRGBEstimation structure** contains results of the estimation:

```
struct LivenessOneShotRGBEstimation {
    enum class State {
        Alive = 0,   //!< The person on image is real
        Fake,        //!< The person on image is fake (photo, printed image)
        Unknown      //!< The liveness status of person on image is Unknown
    };

    float score;        //!< Estimation score
    State state;        //!< Liveness status
    float qualityScore; //!< Liveness quality score
};
```

Estimation score is normalized in range [0..1], where 1 - is real person, 0 - is fake.

Liveness quality score is an image quality estimation for the liveness recognition.

This parameter is used for filtering if it is possible to make bestshot when checking for liveness.

The reference score is 0,5.

The value of `State` depends on `score` and `qualityThreshold`. The value `qualityThreshold` can be given as an argument of method `estimate` (see `ILivenessOneShotRGBEstimator`), and in configuration file *faceengine.conf* (see *ConfigurationGuide* `LivenessOneShotRGBEstimator`).

Recommended thresholds:

Table below contains thresholds from faceengine configuration file (faceengine.conf) in the `LivenessOneShotRGBEs` `::Settings` section. By default, these threshold values are set to optimal.

Table 59: "LivenessOneShotRGB estimator recommended thresholds"

Threshold	Recommended value
realThreshold	0.5
qualityThreshold	0.5
calibrationCoeff	0.947
calibrationCoeff_v12	0.9027

Configurations:

See the "LivenessOneShotRGBEstimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

ILivenessOneShotRGBEstimator

Plan files, version 12

- oneshot_rgb_liveness_v12_model_1_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_2_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_3_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_7_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_1_gpu.plan
- oneshot_rgb_liveness_v12_model_2_gpu.plan
- oneshot_rgb_liveness_v12_model_3_gpu.plan
- oneshot_rgb_liveness_v12_model_7_gpu.plan

Extended plan files, version 12:

- oneshot_rgb_liveness_v12_model_8_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_9_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_10_cpu-avx2.plan
- oneshot_rgb_liveness_v12_model_8_gpu.plan
- oneshot_rgb_liveness_v12_model_9_gpu.plan
- oneshot_rgb_liveness_v12_model_10_gpu.plan

Plan files, version 13

- oneshot_rgb_liveness_v13_model_1_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_2_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_3_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_7_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_1_gpu.plan

- oneshot_rgb_liveness_v13_model_2_gpu.plan
- oneshot_rgb_liveness_v13_model_3_gpu.plan
- oneshot_rgb_liveness_v13_model_7_gpu.plan

Extended plan files, version 13:

- oneshot_rgb_liveness_v13_model_8_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_9_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_10_cpu-avx2.plan
- oneshot_rgb_liveness_v13_model_8_gpu.plan
- oneshot_rgb_liveness_v13_model_9_gpu.plan
- oneshot_rgb_liveness_v13_model_10_gpu.plan

6.20.6.1 Usage example

The face in the image and the image itself should meet the estimator requirements.

You can find additional information in example (`examples/example_estimation/main.cpp`) or in the code below.

```
// Minimum detection size in pixels.
constexpr int minDetSize = 200;

// Step back from the borders.
constexpr int borderDistance = 10;

if (std::min(detectionRect.width, detectionRect.height) < minDetSize) {
    std::cerr << "Bounding Box width and/or height is less than `minDetSize`
        - " << minDetSize << std::endl;
    return false;
}

if ((detectionRect.x + detectionRect.width) > (image.getWidth() -
    borderDistance) || detectionRect.x < borderDistance) {
    std::cerr << "Bounding Box width is out of border distance - " <<
        borderDistance << std::endl;
    return false;
}

if ((detectionRect.y + detectionRect.height) > (image.getHeight() -
    borderDistance) || detectionRect.y < borderDistance) {
    std::cerr << "Bounding Box height is out of border distance - " <<
        borderDistance << std::endl;
    return false;
}
```

```
// Yaw, pitch and roll.
constexpr int principalAxes = 20;

if (std::abs(headPose.pitch) > principalAxes ||
    std::abs(headPose.yaw) > principalAxes ||
    std::abs(headPose.roll) > principalAxes ) {

    std::cerr << "Can't estimate LivenessOneShotRGBEstimation. " <<
        "Yaw, pith or roll absolute value is larger than expected value: "
           << principalAxes << "." <<
        "\nPitch angle estimation: " << headPose.pitch <<
        "\nYaw angle estimation: " << headPose.yaw <<
        "\nRoll angle estimation: " << headPose.roll << std::endl;
    return false;
}
```

> We recommend using `Detector type 3 (fsdk::ObjectDetectorClassType::FACE_DET_V3)`.

6.20.7 NIR Liveness estimation

Name: NIRLivenessEstimator

Algorithm description:

The estimator determines whether a person's face is real or a fake representation, such as a photo or printed image. This estimator relies on images captured by an infrared camera and provides a boolean output indicating whether the face is real (`true`) or fake (`false`).

Implementation description:

The estimator (see `INIRLivenessEstimator` in `INIRLivenessEstimator.h`) implements the `estimate()` function, which accepts a **source warped image** (see the chapter "Image Warping" for details) in R16 format, along with the `fsdk::NIRLivenessEstimation` structure. This method evaluates whether the face in the input image corresponds to a real person. The output of the estimation is a floating-point score normalized in the range [0..1], where a score of 1 indicates a real person and a score of 0 indicates a fake representation.

The **NIRLivenessEstimation structure** contains results of the estimation:

```
struct NIRLivenessEstimation {
     enum class State {
         Real = 0,
         Fake = 1,
         Unknown
    };

    float score;
    State state;
};
```

Recommended thresholds:

Table below contains a threshold from FaceEngine configuration file (faceengine.conf) in `NIRLivenessEstimator::Settings` section. By default, the threshold value is set to optimal.

Table 60: "NIRLivenessEstimator estimator recommended threshold"

Threshold	Description	Recommended value
`realThreshold`	Threshold in [0..1] range.	0.5

Configurations:

See the “NIRLivenessEstimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

INIRLivenessEstimator

.plan files:

- nir_liveness_v2_model_1_cpu.plan
- nir_liveness_v2_model_1_gpu.plan
- nir_liveness_v3_model_2_cpu.plan
- nir_liveness_v3_model_2_gpu.plan

6.21 Personal Protection Equipment Estimation

Name: PPEEstimator

Algorithm description:

The Personal Protection Equipment (PPE) estimator predicts whether a person is wearing one or multiple types of protection equipment, such as:

- Helmet
- Hood
- Vest
- Gloves
- Safety harness

For each attribute, the estimator returns 3 prediction scores which indicate the possibility of person wearing that attribute, not wearing it, and an "unknown" score which will be the highest of them all, if the estimator wasn't able to tell whether a person in the image is wearing a particular attribute.

To correctly determine a personal protective equipment, the following requirements must be met:

- Scene requirements:
  - Moving objects must be visually separated from each other in the image.
  - A background must be mostly static and must not change rapidly.
  - Maximum image shifts due to camera shakes is 1% of the frame size.
  - Overlapping of moving objects by static objects, such as columns, industrial items, and so on, must be minimal.
  - The analyzed scene must not have reflective surfaces. If any, they need to be disguised.
  - Large obstacles should be avoided in the camera's field of view. Pillars, tower cranes, stacked materials, and so on will cause tracks to break and also overlap people. If it is impossible, we recommend that you do not place an obstacle in the center of the frame.
  - Strong camera lights are allowed in a frame. We do not recommend that you point the camera at spotlights and active welding zones, especially in the foreground, because it reduces the visibility of people and the visibility of PPE on them.
  - The camera lens should be kept clean and free of dust. We do not recommend that you place cameras above a material unloading area or near ventilation shafts, because dust on the lens reduces the visibility of people and the visibility of PPE on them.
  - Shooting angle must be without tilting the camera too much. From a top-down perspective, PPE (vest and gloves) can be less visible.
- Image requirements:
  - A person and PPE must be clearly visible to the human eye.
  - Overlapping of a person or PPE with an obstacle or another person and cropping by frame boundaries should not exceed 25%.
  - The linear dimensions of PPE should not exceed 65% of the corresponding frame size.

- The image must not be noisy or distorted by compression algorithm artifacts. The image must be a color one.
- The duration of visibility of a PPE must be at least 10-13 frames.
- The height of the image of a person in pixels must be not less than 100. The minimum pixel density per meter (height of the object in pixels to the height of the object in meters) is 60ppm.
- The minimum height and color of an equipment on body parts must be as follows:

Equipment	Minimum hight, in pixels	Color
Vest	50	Light green (green), yellow, orange
Helmet	20	White, yellow, orange, red
Hood	20	N/A
Gloves	20	White, gray, black
Safety harness	50	N/A

- Video stream requirements:

Parameter	Requirement
Minimum resolution	640x360 pixels
Maximum resolution	1920x1080 pixels
Minimum frame frequency	13 frames per second

- Lighting requirements:

Parameter	Requirement
Scene lighting	200 lux or more
Sudden changes in lighting	None

Implementation description:

The **Personal Protection Equipment Estimation structure** for each attribute looks as follows:

```
struct OnePPEEstimation {
    float positive = 0.0f;
    float negative = 0.0f;
```

```
        float unknown  = 0.0f;

        enum class PPEState : uint8_t {
            Positive, //!< person is wearing specific personal equipment;
            Negative, //!< person isn't wearing specific personal equipment;
            Unknown,  //!< it's hard to tell wether person wears specific
                personal equipment.
            Count     //!< state count
        };

        /**
         * @brief returns predominant personal equipment state
         * */
        inline PPEState getPredominantState();
    };
```

> All three prediction scores sum up to 1.

> The estimator takes an image and a human bounding box of a person for which attributes shall be predicted as an input. For more information about human detector, see “Human Detection” section.

API structure name:

IPPEEstimator

Plan files:

- ppe_estimation_v3_cpu.plan
- ppe_estimation_v3_cpu-avx2.plan
- ppe_estimation_v3_gpu.plan

6.22 Medical Mask Estimation Functionality

Name: MedicalMaskEstimator

This estimator aims to detect a medical mask on the face in the source image. For the interface with MedicalMaskEstimation it can return the next results:

- A medical mask is on the face (see MedicalMask::Mask field in the MedicalMask enum);
- There is no medical mask on the face (see MedicalMask::NoMask field in the MedicalMask enum);
- The face is occluded with something (see MedicalMask::OccludedFace field in the MedicalMask enum);

For the interface with MedicalMaskEstimationExtended it can return the next results:

- A medical mask is on the face (see MedicalMaskExtended::Mask field in the MedicalMaskExtended enum);
- There is no medical mask on the face (see MedicalMaskExtended::NoMask field in the MedicalMaskExtended enum);
- A medical mask is not on the right place (see MedicalMaskExtended::MaskNotInPlace field in the MedicalMaskExtended enum);
- The face is occluded with something (see MedicalMaskExtended::OccludedFace field in the MedicalMaskExtended enum);

The estimator (see IMedicalMaskEstimator in IEstimator.h):

- Implements the *estimate()* function that accepts source warped image in R8G8B8 format and medical mask estimation structure to return results of estimation;
- Implements the *estimate()* function that accepts source image in R8G8B8 format, face detection to estimate and medical mask estimation structure to return results of estimation;
- Implements the *estimate()* function that accepts fsdk::Span of the source warped images in R8G8B8 format and fsdk::Span of the medical mask estimation structures to return results of estimation;
- Implements the *estimate()* function that accepts fsdk::Span of the source images in R8G8B8 format, fsdk::Span of face detections and fsdk::Span of the medical mask estimation structures to return results of the estimation.

Every method can be used with MedicalMaskEstimation and MedicalMaskEstimationExtended.

The estimator was implemented for two use-cases:

1. When the user already has warped images. For example, when the medical mask estimation is performed right before (or after) the face recognition.
2. When the user has face detections only.

Note: Calling the *estimate()* method with warped image and the *estimate()* method with image and detection for the same image and the same face could lead to different results.

6.22.1 MedicalMaskEstimator thresholds

The estimator returns several scores, one for each possible result. The final result is based on that scores and thresholds. If some score is above the corresponding threshold, that result is estimated as final. If none of the scores exceed the matching threshold, the maximum value will be taken. If some of the scores exceed their thresholds, the results will take precedence in the following order for the case with MedicalMaskEstimation:

```
Mask, NoMask, OccludedFace
```

and for the case with MedicalMaskEstimationExtended:

```
Mask, NoMask, MaskNotInPlace, OccludedFace
```

The default values for all thresholds are taken from the configuration file. See Configuration guide for details.

6.22.2 MedicalMask enumeration

The MedicalMask enumeration contains all possible results of the MedicalMask estimation:

```
    enum class MedicalMask {
        Mask = 0,                   //!< medical mask is on the face
        NoMask,                     //!< no medical mask on the face
        OccludedFace                //!< face is occluded by something
    };

    enum class DetailedMaskType {
        CorrectMask = 0,                 //!< correct mask on the face (mouth
           and nose are covered correctly)
        MouthCoveredWithMask,            //!< mask covers only a mouth
        ClearFace,                       //!< clear face - no mask on the face
        ClearFaceWithMaskUnderChin,      //!< clear face with a mask around of
            a chin, mask does not cover anything in the face region (from
           mouth to eyes)
        PartlyCoveredFace,               //!< face is covered with not a
           medical mask or a full mask
        FullMask,                        //!< face is covered with a full mask
            (such as balaclava, sky mask, etc.)
        Count
    };
```

- `Mask` is according to `CorrectMask` or `MouthCoveredWithMask`;
- `NoMask` is according to `ClearFace` or `ClearFaceWithMaskUnderChin`;
- `OccludedFace` is according to `PartlyCoveredFace` or `FullMask`.

Note - `NoMask` means absence of medical mask or any occlusion in the face region (from mouth to eyes).
Note - `DetailedMaskType` is not supported for NPU-based platforms.

6.22.3 MedicalMaskEstimation structure

The `MedicalMaskEstimation` structure contains results of the estimation:

```
struct MedicalMaskEstimation {
    MedicalMask result;           //!< estimation result (@see
        MedicalMask enum)
    DetailedMaskType maskType;    //!< detailed type  (@see
        DetailedMaskType enum)

    // scores
    float maskScore;         //!< medical mask is on the face score
    float noMaskScore;       //!< no medical mask on the face score
    float occludedFaceScore; //!< face is occluded by something score

    float scores[static_cast<int>(DetailedMaskType::Count)]{};    //!<
        detailed estimation scores

    inline float getScore(DetailedMaskType type) const;
};
```

There are two groups of the fields:

1⃣ The first group contains the result:

```
MedicalMask result;
```

Result enum field MedicalMaskEstimation contains the target results of the estimation. Also you can see the more detailed type in MedicalMaskEstimation.

```
DetailedMaskType maskType;           //!< detailed type
```

2⃣ The second group contains scores:

```
float maskScore;           //!< medical mask is on the face score
```

```
    float noMaskScore;          //!< no medical mask on the face score
    float occludedFaceScore;  //!< face is occluded by something score
```

The score group contains the estimation scores for each possible result of the estimation. All scores are defined in [0,1] range. They can be useful for users who want to change the default thresholds for this estimator. If the default thresholds are used, the group with scores could be just ignored in the user code. More detailed scores for every type of a detailed type of face covering are

```
float scores[static_cast<int>(DetailedMaskType::Count)]{};    //!< detailed
    estimation scores
```

- `maskScore` is the sum of scores for `CorrectMask`, `MouthCoveredWithMask`;
- `NoMask` is the sum of scores for `ClearFace` and `ClearFaceWithMaskUnderChin`;
- `occludedFaceScore` is the sum of scores for `PartlyCoveredFace` and `FullMask` fields.

Note - `DetailedMaskType`, `scores`, `getScore` are not supported for NPU-based platforms. It means a user cannot use this fields and methods in code.

6.22.4 MedicalMaskExtended enumeration

The MedicalMask enumeration contains all possible results of the MedicalMask estimation:

```
    enum class MedicalMaskExtended {
        Mask = 0,                   //!< medical mask is on the face
        NoMask,                     //!< no medical mask on the face
        MaskNotInPlace,             //!< mask is not on the right place
        OccludedFace                //!< face is occluded by something
    };
```

6.22.5 MedicalMaskEstimationExtended structure

The MedicalMaskEstimationExtended structure contains results of the estimation:

```
    struct MedicalMaskEstimationExtended {
        MedicalMaskExtended result;     //!< estimation result (@see
            MedicalMaskExtended enum)
        // scores
        float maskScore;          //!< medical mask is on the face score
        float noMaskScore;        //!< no medical mask on the face score
        float maskNotInPlace;     //!< mask is not on the right place
        float occludedFaceScore; //!< face is occluded by something score
```

```
    };
```

There are two groups of the fields:

1⌧ The first group contains only the result enum:

```
        MedicalMaskExtended result;
```

Result enum field MedicalMaskEstimationExtended contains the target results of the estimation.

2⌧ The second group contains scores:

```
        float maskScore;          //!< medical mask is on the face score
        float noMaskScore;        //!< no medical mask on the face score
        float maskNotInPlace;     //!< mask is not on the right place
        float occludedFaceScore; //!< face is occluded by something score
```

The score group contains the estimation scores for each possible result of the estimation. All scores are defined in [0,1] range.

6.22.6 Filtration parameters

The estimator is trained to work with face images that meet the following requirements:

Table 64: "Requirements for `fsdk::MedicalMaskEstimator::EstimationResult`"

Attribute	Acceptable values
headPose.pitch	[-40…40]
headPose.yaw	[-40…40]
headPose.roll	[-40…40]
ags	[0.5…1.0]

Configurations:

See the "Medical mask estimator settings" section in the "ConfigurationGuide.pdf" document.

API structure name:

IMedicalMaskEstimator

Plan files:

- mask_clf_v3_cpu-avx2-int8.plan

- mask_clf_v3_gpu.plan
- mask_clf_v3_gpu-fp16.plan

6.23 Human Attribute Estimation

Name: HumanAttributeEstimator

Algorithm description:

This estimator aims to detect next human attributes on the warped human image:

- Age;
- Gender;
- Sleeve size;
- The presence of a headwear;
- The color of a headwear;
- The presence of a backpack;
- Estimation of the lower body clothing type;
- The color of a lower body clothing;
- Outwear color.
- The color of the shoes;

Age estimation contains a single number - the number of years.

Gender estimation contains one of the next results (see `HumanAttributeResult::Gender` enum):

- Person's gender is female;
- Person's gender is male;
- Person's gender is unknown.

Sleeve size estimation contains one of the next results (see `HumanAttributeResult::SleeveSize` enum):

- Person's sleeves are short;
- Person's sleeves are long;
- Person's sleeves size is unknown.

Hat estimation contains one of the next results (see `HumanAttributeResult::Hat` enum):

- There is no headwear;
- There is a headwear;
- Headwear state is unknown.

Backpack estimation contains one of the next results (see `HumanAttributeResult::Backpack` enum):

- There is no backpack;
- There is a backpack;
- Backpack state is unknown.

LowerBodyClothing estimation contains one of the next results (see `HumanAttributeResult::LowerBodyClothing` enum):

- There are pants;
- There are shorts;
- There is skirt;
- Lower body clothing state is unknown.

Outwear color estimation contains the next results (see `HumanAttributeResult::Color` enum):

- Person's outwear color is black;
- Person's outwear color is blue;
- Person's outwear color is green;
- Person's outwear color is grey;
- Person's outwear color is orange;
- Person's outwear color is purple;
- Person's outwear color is red;
- Person's outwear color is white;
- Person's outwear color is yellow;
- Person's outwear color is pink;
- Person's outwear color is brown;
- Person's outwear color is beige;
- Person's outwear color is khaki;
- Person's outwear color is multicolored.

Apparent color estimation contains the next results (see `HumanAttributeResult::ApparentColor` enum):

- Apparent color is black;
- Apparent color is white;
- Apparent color is some other color from full palette;
- Apparent color is unknown.

Outwear color vs Apparent color:

For now, we have two color palettes Outwear color and Apparent color. Outwear color palette represents full palette supported by human attributes estimator. Apparent color palette is simplified version of Outwear color. Color of some attributes can be classified only of small pool of colors - `Black` and `White` for now. So, in sake of simplification for the user we introduce Apparent color palette. Apparent color palette can be extended with colors in the future.

Implementation description:

The **Gender enumeration** contains all possible results of the Gender estimation:

```
enum class Gender {
    Female,     //!< person's gender is female
    Male,       //!< person's gender is male
```

```
    Unknown     //!< person's gender is unknown
};
```

The **SleeveSize enumeration** contains all possible results of the SleeveSize estimation:

```
enum class SleeveSize {
    Short,   //!< sleeves are short
    Long,    //!< sleeves are long
    Unknown  //!< sleeves state is unknown
};
```

The **Hat enumeration** contains all possible results of the Hat estimation:

```
enum class Hat {
    No,      //< there is no headwear
    Yes,     //< there is a headwear
    Unknown  //< headwear state is unknown
};
```

The **Backpack enumeration** contains all possible results of the Backpack estimation:

```
enum class Backpack {
    No,      //< there is no backpack
    Yes,     //< there is a backpack
    Unknown  //< backpack state is unknown
};
```

The **LowerBodyClothing enumeration** contains all possible results of the LowerBodyClothing estimation:

```
enum class LowerBodyClothing {
    Pants,    //< there is pants
    Shorts,   //< there is shorts
    Skirt,    //< there is skirt
    Unknown   //< lower body clothing state is unknown
};
```

The **Color enumeration** contains all possible results of the OutwearColor estimation:

```
enum class Color {
    Black,
    Blue,
```

```
        Green,
        Grey,
        Orange,
        Purple,
        Red,
        White,
        Yellow,
        Pink,
        Brown,
        Beige,
        Khaki,
        Multicolored,
        Count
    };
```

The **ApparentColor enumeration** contains all possible results of the ApparentColor estimation:

```
    enum class ApparentColor {
        Black,
        White,
        Other,
        Unknown,
        Count
    };
```

Human Attribute estimation request:

HumanAttributeRequest lists all possible estimation attributes that HumanAttributeEstimator is currently able to estimate.

```
    enum class HumanAttributeRequest {
        EstimateAge                 = 1 << 0, //!< estimate age
        EstimateGender              = 1 << 1, //!< estimate gender
        EstimateSleeveSize          = 1 << 2, //!< estimate sleeves size
        EstimateBackpack            = 1 << 3, //!< estimate backpack state
        EstimateOutwearColor        = 1 << 4, //!< estimate outwear color
        EstimateHeadwear            = 1 << 5, //!< estimate headwear state
        EstimateLowerBodyClothing = 1 << 7, //!< estimate lower body
            clothing state
        EstimateShoeColor           = 1 << 8, //!< estimate shoe color
        EstimateAll                 = 0xffff  //!< estimate all attributes
    };
```

The **GenderEstimation structure** contains results of the gender estimation:

```
        struct GenderEstimation {
            Gender result;   //!< estimation result (@see Gender enum).
            float female;    //!< female gender probability score
            float male;      //!< male gender probability score
            float unknown;   //!< unknown gender probability score
        };
```

1⌧ The first group contains only the result enum:

```
            Gender result;   //!< estimation result (@see Gender enum).
```

Result enum field GenderEstimation contain the target results of the estimation.

2⌧ The second group contains scores:

```
            float female;    //!< female gender probability score
            float male;      //!< male gender probability score
            float unknown;   //!< unknown gender probability score
```

The scores group contains the estimation score.

The **SleeveSizeEstimation structure** contains results of the sleeves size estimation:

```
        struct SleeveSizeEstimation {
            SleeveSize result; //!< estimation result (@see SleeveSize enum).
            float shortSize;   //!< short sleeves size probability score
            float longSize;    //!< long sleeves size probability score
            float unknown;     //!< unknown sleeves size probability score
        };
```

1⌧ The first group contains only the result enum:

```
            SleeveSize result; //!< estimation result (@see SleeveSize enum).
```

Result enum field SleeveSizeEstimation contain the target results of the estimation.

2⌧ The second group contains scores:

```
            float shortSize;   //!< short sleeves size probability score
            float longSize;    //!< long sleeves size probability score
            float unknown;     //!< unknown sleeves size probability score
```

The scores group contains the estimation score.

The **HatEstimation structure** contains results of the hat estimation:

```
        struct HatEstimation {
            Hat result;     //!< estimation result (@see Hat enum).
            float noHat;    //!< no hat probability score
            float hat;      //!< hat probability score
            float unknown;  //!< unknown hat state probability score

            ApparentColorEstimation hatColor; //!< hat color estimation
        };
```

1) The first group contains only the result enum:

```
            Hat result;     //!< estimation result (@see Hat enum).
```

Result enum field HatEstimation contain the target results of the estimation.

2) The second group contains scores:

```
            float noHat;    //!< no hat probability score
            float hat;      //!< hat probability score
            float unknown;  //!< unknown hat state probability score
```

The scores group contains the estimation score.

3) The third group contains color estimation:

```
            ApparentColorEstimation hatColor; //!< hat color estimation.
```

The **BackpackEstimation structure** contains results of the backpack estimation:

```
        struct BackpackEstimation {
            Backpack result;   //!< estimation result (@see Backpack enum).
            float noBackpack; //!< no backpack probability score
            float backpack;    //!< backpack probability score
            float unknown;     //!< unknown backpack state probability score
        };
```

1) The first group contains only the result enum:

```
            Backpack result;   //!< estimation result (@see Backpack enum).
```

Result enum field BackpackEstimation contain the target results of the estimation.

2⊠ The second group contains scores:

```
        float noBackpack; //!< no backpack probability score
        float backpack;   //!< backpack probability score
        float unknown;    //!< unknown backpack state probability score
```

The scores group contains the estimation score.

The **LowerBodyClothingEstimation structure** contains results of the lower body clothing estimation:

```
    struct LowerBodyClothingEstimation {
        LowerBodyClothing result;  //!< estimation result.
        float pants;               //!< pants probability score
        float shorts;              //!< shorts probability score
        float skirt;               //!< skirt probability score
        float unknown;             //!< unknown state probability score

        OutwearColorEstimation lowerBodyClothingColor; //!< lower body
           clothing color estimation.
    };
```

1⊠ The first group contains only the result enum:

```
        LowerBodyClothing result;  //!< estimation result.
```

Result enum field LowerBodyClothingEstimation contain the target results of the estimation.

2⊠ The second group contains scores:

```
        float pants;               //!< pants probability score
        float shorts;              //!< shorts probability score
        float skirt;               //!< skirt probability score
        float unknown;             //!< unknown state probability score
```

The scores group contains the estimation score.

3⊠ The third group contains color estimation:

```
        OutwearColorEstimation lowerBodyClothingColor; //!< lower body
           clothing color estimation.
```

The **OutwearColorEstimation structure** contains results of outwear color estimation:

```
struct OutwearColorEstimation {
    bool isBlack;                                   //!< outwear is black
    bool isBlue;                                    //!< outwear is blue
    bool isGreen;                                   //!< outwear is green
    bool isGrey;                                    //!< outwear is grey
    bool isOrange;                                  //!< outwear is orange
    bool isPurple;                                  //!< outwear is purple
    bool isRed;                                     //!< outwear is red
    bool isWhite;                                   //!< outwear is white
    bool isYellow;                                  //!< outwear is yellow
    bool isPink;                                    //!< outwear is pink
    bool isBrown;                                   //!< outwear is brown
    bool isBeige;                                   //!< outwear is beige
    bool isKhaki;                                   //!< outwear is khaki
    bool isMulticolored;                            //!< outwear is
       multicolored
    float scores[static_cast<int>(Color::Count)]; //!< estimation scores

    /**
     * @brief Returns score of required outwear color.
     * @param [in] color outwear color.
     * @see Color for more info.
     * */
    inline float getScore(Color color) const;
};
```

1⃣ The first group contains plain answer:

```
    bool isBlack;                                   //!< outwear is black
    bool isBlue;                                    //!< outwear is blue
    bool isGreen;                                   //!< outwear is green
    bool isGrey;                                    //!< outwear is grey
    bool isOrange;                                  //!< outwear is orange
    bool isPurple;                                  //!< outwear is purple
    bool isRed;                                     //!< outwear is red
    bool isWhite;                                   //!< outwear is white
    bool isYellow;                                  //!< outwear is yellow
    bool isPink;                                    //!< outwear is pink
    bool isBrown;                                   //!< outwear is brown
    bool isBeige;                                   //!< outwear is beige
    bool isKhaki;                                   //!< outwear is khaki
    bool isMulticolored;                            //!< outwear is
       multicolored
```

2⊠ The second group contains scores:

```
        float scores[static_cast<int>(Color::Count)]; //!< estimation scores
```

Note Not all color flags and according float scores in OutwearColorEstimation have valid values. Some colors were added to interface to support future colors expansion and will store valid values as algorithm will evolve release by release. Currently, Pink, Beige, Khaki and Multicolored are zeroed internally.

The **ApparentColorEstimation structure** contains results of apparent color estimation:

```
    struct ApparentColorEstimation {
        bool isBlack;                                          //!<
           attribute is black
        bool isWhite;                                          //!<
           attribute is white
        bool isOther;                                          //!<
           attribute is some other
        bool isUnknown;                                        //!<
           attribute is unknown
        float scores[static_cast<int>(ApparentColor::Count)];  //!<
           estimation scores

        /**
         * @brief Returns score of required color.
         * @param [in] color color.
         * @see ApparentColor for more info.
         * */
        inline float getScore(ApparentColor color) const;
    };
```

1⊠ The first group contains plain answer:

```
        bool isBlack;                                          //!<
           attribute is black
        bool isWhite;                                          //!<
           attribute is white
        bool isOther;                                          //!<
           attribute is some other
        bool isUnknown;                                        //!<
           attribute is unknown
```

2⊠ The second group contains scores:

```
        float scores[static_cast<int>(ApparentColor::Count)];   //!<
            estimation scores
```

The **HumanAttributeResult structure** contains optional results of all estimations depending on HumanAttributeRequest.

```
        /**
         * @brief Age estimation by human body.
         * @note This estimation may be very different from estimation by
             face.
         * */
        Optional<float> age;
        /**
         * @brief Gender estimation by human body.
         * @note This estimation may be very different from estimation by
             face.
         * */
        Optional<GenderEstimation> gender;
        Optional<SleeveSizeEstimation> sleeve;                      //!<
            sleeve estimation.
        Optional<HatEstimation> headwear;                           //!<
            headwear estimation.
        Optional<BackpackEstimation> backpack;                      //!<
            backpack estimation.
        Optional<OutwearColorEstimation> outwearColor;              //!<
            outwear color estimation.
        Optional<LowerBodyClothingEstimation> lowerBodyClothing;    //!<
            lower body clothing estimation.
        Optional<ApparentColorEstimation> shoeColor;                //!<
            shoe color estimation.
```

HumanAttribute Aggregation:

The HumanAttribute provides a method to aggregate output results of a batch estimate call. All valid features are counted and the result is a mean of them. Invalid fields will be skipped and do not influence on aggregation result.

```
        /**
         * @brief Aggregate human body attributes.
         * @details All invalid fields will be skipped and do not influence
             on aggregation result
         * @param [in] estimations span of estimation results.
         * @param [in] request estimation request.
```

```
         * @param [out] result aggregated result.
         * @return Result with error code.
         * @see Span, HumanAttributeResult, IHumanAttributeEstimator::
           EstimationRequest, Result and FSDKError for details.
         * @note all spans should be based on user owned continuous
           collections.
         * @note all spans should be equal size.
         * */
        virtual Result<FSDKError> aggregate(
            Span<const HumanAttributeResult> estimations,
            HumanAttributeRequest request,
            HumanAttributeResult& result) const noexcept  = 0;
```

Attribute dependencies:

Some attribute results are influenced by the outcomes of other attributes. Specifically, the color flag and score of an attribute depend on its predicted type. For example, it is not meaningful to assign color values to an attribute classified as `Unknown`. These rules also apply to aggregation results.

Dependency rules:

- In the `HatEstimation` struct:
  - The `hatColor` field depends on the `result` field.
  - If the `result` field has value `No` or `Unknown`, `hatColor` will be set to `isUnknown = true`, and all scores will be reset to zero.
- In the `LowerBodyClothingEstimation` struct:
  - The `lowerBodyClothingColor` field depends on the `result` field.
  - If the `result` field has value `Unknown`, all flags in `lowerBodyClothingColor` will be set to `false`, and all scores will be reset to zero.
- In the `HumanAttributeResult` struct:
  - The `shoeColor` field depends on the `result` field of `LowerBodyClothingEstimation`.
  - If the `result` field of `LowerBodyClothingEstimation` is `Unknown`, then `shoeColor` will be set to `isUnknown = true`, and all scores will be reset to zero.

Recommended thresholds:

Human Attribute estimator sets outwear color bool values and age by comparing an output score with a corresponding threshold value listed in faceengine.conf file in `HumanAttributeEstimator::Settings` section. By default, these threshold values are set to optimal.

Table 65: "Human Attribute Estimator recommended thresholds"

Thresholds	Recommended values
blackUpperThreshold	0.740

Thresholds	Recommended values
blueUpperThreshold	0.655
brownUpperThreshold	0.985
greenUpperThreshold	0.700
greyUpperThreshold	0.710
orangeUpperThreshold	0.420
purpleUpperThreshold	0.650
redUpperThreshold	0.600
whiteUpperThreshold	0.820
yellowUpperThreshold	0.670
blackLowerThreshold	0.700
blueLowerThreshold	0.840
brownLowerThreshold	0.850
greenLowerThreshold	0.700
greyLowerThreshold	0.690
orangeLowerThreshold	0.760
purpleLowerThreshold	0.890
redLowerThreshold	0.600
whiteLowerThreshold	0.540
yellowLowerThreshold	0.930
adultThreshold	0.940

Configurations:

See the “Human Attribute Estimator settings” section in the “ConfigurationGuide.pdf” document.

API structure name:

IHumanAttributeEstimator

Plan files:

- human_attributes_v2_cpu.plan
- human_attributes_v2_cpu-avx2.plan
- human_attributes_v2_gpu.plan

6.24 Crowd Estimation

Name: CrowdEstimator

Algorithm description:

This estimator aims to count a humans (heads) in the input image. It returns a count and center coordinates of heads (optional).

There are several possible CrowdEstimator work modes:

- Single network - Crowd estimation network is used. It works good with small heads in the image, but can lose big heads (which are closer to the camera).
- Two networks mode - two networks are be used: Crowd estimation with HumanDetector or Crowd estimation with HeadDetector. This mode causes more accurate results, but the execution of the algorithms takes more time. Two variants of detector are possible. They are "HumanDetector" and "HeadDetector". User can change the `detectorType` parameter in the config.

Implementation description:

The estimator (see ICrowdEstimator in ICrowdEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format, the region of interest (ROI), `fsdk::ICrowdEstimator::EstimationRequest` structure and returns the estimation result;
- Implements the *estimate()* function that accepts `fsdk::Span` of the **source images** in R8G8B8 format, `fsdk::Span` of ROIs, `fsdk::ICrowdEstimator::EstimationRequest` structure and `fsdk::Span` of the `fsdk::CrowdEstimation` structures to return results of estimation.

User is free to choose an estimation type. For this purpose, *estimate()* method takes one of the estimation requests:

- `fsdk::ICrowdEstimator::EstimationRequest::estimateHeadCount` to return people (heads) count only;
- `fsdk::ICrowdEstimator::EstimationRequest::estimateHeadCountAndCoords` to return people (heads) count as well as head center coordinates;

The **CrowdEstimation structure** contains all possible results of the Crowd estimation:

```
struct CrowdEstimation {
    size_t count; //!< The number of people (heads) in the image.
    IPointBatchPtr points; //!< Coordinates of people heads. Empty if
        not requested.
};
```

minHeadSize

This estimator can estimate heads with size 3 px and more. In case when such small heads are not required (or not possible in the use-case), user can change the `minHeadSize` parameter in the config.

Before processing, the images will be resized by `minHeadSize/3` times. For example, if the value is `minHeadSize=12`, then the image will be additionally resized by `minHeadSize=12/3=4` times.

Estimator works faster with larger value of `minHeadSize`.

CrowdEstimatorType

The **CrowdEstimation CrowdEstimatorType** contains all possible working modes of the Crowd estimator:

```
enum CrowdEstimatorType {
    CET_DEFAULT = 0,          //!< Default type which is specified in
       config file. @see ISettingsProvider
    CET_SINGLE_NET = 1,       //!< Single network mode - only Crowd
       estimation will be used
    CET_TWO_NETS = 2,         //!< Double network mode - Crowd +
       HeadDetector
    CET_COUNT
};
```

Here are:

- CET_DEFAULT - the default mode which is recommended to use. The result working mode will be determines by the value in the configuration file faceengine.conf.
- CET_SINGLE_NET - single network working mode. Only Crowd estimation will be used.
- CET_TWO_NETS - two networks mode: Crowd estimation and HumanDetector or Crowd estimation and HeadDetector.
- CET_COUNT - just a stub to check an input correctness, do not use it.

API structure name:

ICrowdEstimator

Plan files:

- crowd_v2_cpu.plan
- crowd_v2_cpu-avx2.plan
- crowd_v2_gpu.plan

6.25 Fights Estimation

Name: FightsEstimator

Algorithm description:

This estimator detects fights on a video by processing several images sequences (batches) one by one from the target video.

This estimator works based on the several image sequences (batches). Each batch should contain the `IFightsEstimator::getBatchSize()` frames.

Every `IFightsEstimator::estimate` estimation call returns a context structure as a result. This context structure should be passed to the next estimation call for the current video. If several videos should be processed in parallel, you should keep different context structures - one for each video. For the first estimation call, the context structure should be empty (`nullptr`). After estimating the `IFightsEstimator::getMinBatchCount()` batches, the context structure will contain `IFightsEstimatorContext::State::Ready`. You can then take an estimation result by calling the `IFightsEstimatorContext::getResult()` method. If more frames should be processed, the succeeding `IFightsEstimator::estimate` calls are required with passing the context structure.

Input requirements:

- Frames should be in the `fsdk::Format::R8B8G8` format.
- Video should be about 30 FPS.
  If the video contains more FPS (for example, 60 FPS), we recommend that you do not pass every frame to the estimator (for example, every second frame for the 60 FPS video).

Content requirements:

- Human bounding box heights in the video should be >=30% frames hight.
  For example, for the video with 640 x 480 resolution the minimum humans bounding box height should be (640 * 0.3) = 192 px.
  For details, see the Human Detection section in the Face detection facility chapter.

Camera requirements:

- A camera should be static.
- An RGB camera. The estimator performance on IR cameras is worse.
- The perspective should be from top to bottom, as on CCTV cameras. The recommended range is 30 to 60 degrees. The images below show examples of suitable angles.

Implementation description:

The estimator (see IFightsEstimator in IFightsEstimator.h):

- Implements the `estimate()` function that needs the `fsdk::Span` (batch) of `fsdk::Image` objects and the `fsdk::IFightsEstimatorContextPtr` context object. The result is an error code with updated `fsdk::IFightsEstimatorContextPtr` context object.

The context structure (see IFightsEstimatorContext in IFightsEstimator.h):

- Implements the `getState()` function that takes no arguments. The result is the current estimation state.
  Value `IFightsEstimatorContext::State::Ready` means that the estimation is completed

and the result could be taken from the structure. Value `IFightsEstimatorContext::State::NoReady` means that the estimation requires more frames to proceed.

- Implements the `getResult()` function that takes no arguments. The result is the current estimation result (`FightsEstimation` structure).

The **`FightsEstimation` structure** contains results of the estimation:

```
struct FightsEstimation {
    enum class State {
        NoFight, //!< There is no fight on the input frames
        Fight    //!< Fight detected on the input frames
    };
    State state; //!< Estimation status
    float score; //!< Estimation score normalized to [0..1] range
};
```

Estimation score is normalized in range [0..1], where 1 - is a real person, 0 - is a fake.

The value of `state` depends on `threshold`. You can change the `threshold` value in the *faceengine.conf* configuration file. For details, see the FightsEstimator settings section in Configuration Guide.

API structure name:

IFightsEstimator

Plan files:

- fights_v2_cpu.plan
- fights_v2_cpu-avx2.plan
- fights_v2_gpu.plan

6.26 ImageModification Estimation

Name: ImageModificationEstimator

Algorithm description:

This estimator checks images for several **specific types of modifications**:

1. A logo on top of the image
2. Black border[s] at the edge[s] of the image. Beware, letter boxing and pillar boxing fall into this category as well.
3. Superimposed shapes. The estimator determines whether there are filled shapes on top of the image
4. Multiple overlapping images. The estimator determines if there are two or more different images that partially overlap. Each of the original images might be free from modifications on its own, but they overlap within the final, composed image under estimation.

To recap, the estimator checks images for the traces of most unnatural, artificial ways of image compositing and editing.

The estimator does not analyze images for all possible types of modifications.

For example, the estimator is not intended to find synthetic images generated by DeepFake technologies (we offer DeepFakeEstimator for that).

Similarly, the estimator is not intended to detect presentation attacks (we offer several types of liveness estimators for that).

The estimator returns results as `ImageModificationEstimation` or a span of that type.

`ImageModificationEstimation` contains a confidence score in the range [0.0,1.0] that the image is not modified.

`0` means that the estimator has found the traces of modification, hence the image is modified. `1` means that the estimator has not found the traces of modification of any of the types listed above, hence the image is probably not modified in the ways listed above.

Note: The face in the image and the image itself should meet the estimator requirements: - only 1 face in the image - face size > 50px

Note: This algorithm may produce incorrect results on images with a blurred background.

Implementation description:

The estimator (see IImageModificationEstimator in IImageModificationEstimator.h):

- Implements the *estimate()* function that accepts **source image** in R8G8B8 format and returns the estimation result;

- Implements the *estimate()* function that accepts `fsdk::Span` of the **source images** in R8G8B8 format and `fsdk::Span` of the `fsdk::ImageModificationEstimation` structures to return the results of estimation.
- Implements the *validate()* function that accepts `fsdk::Span` of the **source images** and `fsdk::Span` of the `fsdk::Result<fsdk::FSDKError>` structures to return the results of validation for each image.

The **ImageModificationEstimation structure** contains the estimation results:

```
/**
 * @brief Image modification estimator output structure
 * */
struct ImageModificationEstimation {
    enum class Status {
        Modified = 0, //!< Modification traces are found, the image is
            modified
        Unmodified    //!< Modification traces are not found, the image is
            probably unmodified
    };

    float score; //!< confidence score in the [0,1] range that the image
        does not contain modification traces.
                          //!< "0" means that modification traces are
                              certainly present.

    Status status; //!< whether the image is modified
};
```

The convenience variable `ImageModificationEstimation::status` is computed by comparing `score` with `ImageModificationEstimator::Settings::threshold` from `faceengine.conf`. For details, see the "ImageModificationEstimation settings" section in Configuration Guide.

API structure name:

IImageModificationEstimator

Plan files:

- image_modification_v1_cpu.plan
- image_modification_v1_cpu-avx2.plan
- image_modification_v1_gpu.plan

Examples See

- `example_image_modification` in C++ and
- `example_image_modification.py` in Python

7 Descriptor Processing Facility

7.1 Overview

The section describes descriptors and all the processes and objects corresponding to them.

Descriptor itself is a set of object parameters that are specially encoded. Descriptors are typically more or less invariant to various affine object transformations and slight color variations. This property allows efficient use of such sets to identify, lookup, and compare real-world objects images.

To receive a descriptor you should perform a special operation called descriptor *extraction*.

The general case of descriptors usage is when you compare two descriptors and find their similarity score. Thus you can identify persons by comparing their descriptors with your descriptors database.

All descriptor comparison operations are called *matching*. The result of the two descriptors matching is a distance between components of the corresponding sets that are mentioned above. Thus, from a magnitude of this distance, we can tell if two objects are presumably the same.

There are two different tasks solved using descriptors: person identification and person reidentification.

7.1.1 Person Identification Task

Facial recognition is the task of making an identification of a face in a photo or video image against a pre-existing database of faces. It begins with detection - distinguishing human faces from other objects in the image - and then works on the identification of those detected faces. To solve this problem, we use a face descriptor, which extracted from an image face of a person. A person's face is invariable throughout his life.

In a case of the face descriptor, the extraction is performed from object image areas around some previously discovered facial landmarks, so the quality of the descriptor highly depends on them and the image it was obtained from.

The process of face recognition consists of 4 main stages:

- face detection in an image;
- warping of face detection – compensation of affine angles and centering of a face;
- descriptor extraction;
- comparing of extracted descriptors (matching).

> Additionally you can extract face features (gender, age, emotions, etc) or image attributes (light, dark, blur, specularity, illumination, etc.).

7.1.2 Person Reidentification Task

Note! This functionality is experimental.

The person reidentification enables you to detect a person who appears on different cameras. For example, it is used when you need to track a human, who appears on different supermarket cameras. Reidentification can be used for:

- building of human traffic warm maps;
- analysing of visitors movement across cameras network;
- tracking of visitors across cameras network;
- search for a person across the cameras network in case when face was not captured (e.g. across CCTV cameras in the city);
- etc.

For reidentification purposes, we use so-called human descriptors. The extraction of the human descriptor is performed using the detected area with a person's body on an image or video frame. The descriptor is a unique data set formed based on a person's appearance. Descriptors extracted for the same person in different clothes will be significantly different.

> The face descriptor and the human descriptor are almost the same from the technical point of view, but they solve fundamentally different tasks.

The process of reidentifications consists of the following stages:

- human detection in an image;
- warping of human detection – centering and cropping of the human body;
- descriptor extraction;
- comparing of extracted descriptors (matching).

> The human descriptor does not support the *descriptor score* at all. The returned value of the descriptor score is always equal to 1.0.

The human descriptor is based on to the following criteria:

- clothes (type and color);
- shoes;
- accessories;
- hairstyle;
- body type;
- anthropometric parameters of the body.

Note. The human reidentification algorithm is trained to work with input data that meets the following requirements:

- input images should be in R8G8B8 format (will work worse in night mode);
- the smaller side of input crop should be greater than 60 px;
- inside of same crop, one person should occupy more than 80% (sometimes several persons fit into the same frame).

7.2 Descriptor

Descriptor object stores a compact set of packed properties as well as some helper parameters that were used to extract these properties from the source image. Together these parameters determine descriptor compatibility. Not all descriptors are compatible with each other. It is impossible to batch and match incompatible descriptors, so you should pay attention to what settings do you use when extracting them. Refer to section “Descriptor extraction” for more information on descriptor extraction.

7.2.1 Descriptor Versions

Face descriptor algorithm evolves with time, so newer FaceEngine versions contain improved models of the algorithm.

> Descriptors of different versions are **incompatible**! This means that you **cannot match descriptors with different versions**. This does not apply to base and mobilenet versions of the same model: they are compatible.

See chapter “Appendix A. Specifications” for details about performance and precision of different descriptor versions.

Descriptor version 65 is the most precise one. And it works well with the personal protective equipment on face like medical mask.

Descriptor version may be specified in the configuration file (see section “Configuration data” in chapter “Core facility”).

7.2.1.1 Face descriptor

7.2.1.1.1 Available versions Currently, the following versions are available: 58, 59, 60, 62, and 65. These descriptors have two implementation types:

- **Backend**: High-precision implementation
- **Mobilenet**: Faster implementation with smaller model files

7.2.1.1.2 Version compatibility

- Versions 58, 62, and 65 support only the **backend** implementation.
- Backend versions offer higher precision.
- Mobilenet versions provide faster processing and smaller model files.

For detailed performance and precision comparisons, see Appendix A.1 and A.2.

7.2.1.1.3 GPU compatibility for CNN65 CNN65 requires a GPU with NVIDIA Turing architecture or newer for correct operation. Older GPU architectures are not supported and may experience performance issues or complete failure.

7.2.1.2 Human descriptor

Versions of human descriptors are available: 102, 103, 104, 105, 106, 107, 108, 109, 110, 112, 113, 115, 116

> Versions 102, 103, 104, 105, 106, 107, 109, 110 are deprecated.

To create a human descriptor, human batch, human descriptor extractor, human descriptor matcher you must pass the human descriptor version

- DV_MIN_HUMAN_DESCRIPTOR_VERSION = 102 or
- HDV_TRACKER_HUMAN_DESCRIPTOR_VERSION = 102, //!< Deprecated. human descriptor for tracking of people on one camera, light and fast version
- HDV_PRECISE_HUMAN_DESCRIPTOR_VERSION = 103, //!< Deprecated. precise human descriptor, heavy and slow
- HDV_REGULAR_HUMAN_DESCRIPTOR_VERSION = 104, //!< Deprecated. regular human descriptor, use it by default for multi-cameras tracking
- HDV_TRACKER_V2 = 105, //!< human descriptor for tracking of people, light and fast version.
- HDV_PRECISE_V2 = 106, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V2 = 107, //!< regular human descriptor.
- HDV_TRACKER_V3 = 108, //!< human descriptor for tracking of people, light and fast version.
- HDV_PRECISE_V3 = 109, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V3 = 110, //!< regular human descriptor.
- HDV_PRECISE_V4 = 112, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V4 = 113 //!< regular human descriptor.
- HDV_PRECISE_V5 = 115, //!< precise human descriptor, heavy and slow.
- HDV_REGULAR_V5 = 116 //!< regular human descriptor.

7.2.2 Descriptor Batch

When matching significant amounts of descriptors, it is desired that they reside continuously in memory for performance reasons (think cache-friendly data locality and coherence). This is where descriptor batches come into play. While descriptors are optimized for faster creation and destruction, batches are optimized for long life and better descriptor data representation for the hardware.

A batch is created by the factory like any other object. Aside from type, a size of the batch should be specified. Size is a memory reservation this batch makes for its data. It is impossible to add more data than specified by this reservation.

Next, the batch must be populated with data. You have the following options:

- add an existing descriptor to the batch;
- load batch contents from an archive.

The following notes should be kept in mind:

- When adding an existing descriptor, its data is copied into the batch. This means that the descriptor object may be safely released.
- When adding the first descriptor to an empty batch, initial memory allocation occurs. Before that moment the batch does not allocate. At the same moment, internal descriptor helper parameters are copied into the batch (if there are any). This effectively determines compatibility possibilities of the batch. When the batch is initialized, it does not accept incompatible descriptors.

After initialization, a batch may be matched pretty much the same way as a simple descriptor.

Like any other data storage object, a descriptor batch implements the *::clear()* method. An effect of this method is the batch translation to a non-initialized state **except memory deallocation**. In other words, batch capacity stays the same, and no memory is reallocated. However, an actual number of descriptors in the batch and their parameters are reset. This allows re-populating the batch.

Memory deallocation takes place when a batch is released.

Care should be taken when serializing and deserializing batches. When a batch is created, it is assigned with a fixed-size memory buffer. The size of the buffer is embedded into the batch BLOB when it is saved. So, when allocating a batch object for reading the BLOB into, make sure its size is at least the same as it was for the batch saved to the BLOB (even if it was not full at the moment). Otherwise, loading fails. Naturally, it is okay to deserialize a smaller batch into a larger another batch this way.

7.2.3 Descriptor Extraction

Descriptor extractor is the entity responsible for descriptor extraction. Like any other object, it is created by the factory. To extract a descriptor, aside from the source image, you need:

- a face detection area inside the image (see chapter “Detection facility”)
- a pre-allocated descriptor (see section “Descriptor”)
- a pre-computed landmarks (see chapter “Image warping”)

A descriptor extractor object is responsible for this activity. It is represented by the straightforward *IDescriptorExtractor* interface with only one method *extract()*. Note, that the descriptor object must be created prior to calling *extract()* by calling an appropriate factory method.

Landmarks are used as a set of coordinates of object points of interest, that in turn determine source image areas, the descriptor is extracted from. This allows extracting only data that matters most for a particular type of object. For example, for a human face we would want to know at least definitive properties of eyes, nose, and mouth to be able to compare it to another face. Thus, we should first invoke a feature extractor to locate where eyes, nose, and mouth are and put these coordinates into landmarks. Then the descriptor extractor takes those coordinates and builds a descriptor around them.

Descriptor extraction is one of the most computation-heavy operations. For this reason, threading might be considered. Be aware that descriptor extraction is not thread-safe, so you have to create an extractor object per a worker thread.

It should be noted, that the face detection area and the landmarks are required only for image warping, the preparation stage for descriptor extraction (see chapter "Image warping"). If the source image is already warped, it is possible to skip these parameters. For that purpose, the *IDescriptorExtractor* interface provides a special *extractFromWarpedImage()* method.

Descriptor extraction implementation supports execution on GPUs.

The *IDescriptorExtractor* interface provides *extractFromWarpedImageBatch()* method which allows you to extract batch of descriptors from the image array in one call. This method achieve higher utilization of GPU and better performance (see the "GPU mode performance" table in appendix A chapter "Specifications").

Also *IDescriptorExtractor* returns *descriptor score* for each extracted descriptor. Descriptor score is normalized value in range [0,1], where 1 - face in the warp, 0 - no face in the warp. This value allows you filter descriptors extracted from false positive detections.

The *IDescriptorExtractor* interface provides *extractFromWarpedImageBatchAsync()* method which allows you to extract batch of descriptors from the image array asynchronously in one call. This method achieve higher utilization of GPU and better performance (see the "GPU mode performance" table in appendix A chapter "Specifications").

Note: Method *extractFromWarpedImageBatchAsync()* is experimental, and it's interface may be changed in the future.

Note: Method *extractFromWarpedImageBatchAsync()* is not marked as noexcept and may throw an exception.

7.2.4 Descriptor Matching

It is possible to match a pair (or more) previously extracted descriptors to find out their similarity. With this information, it is possible to implement face search and other analysis applications.

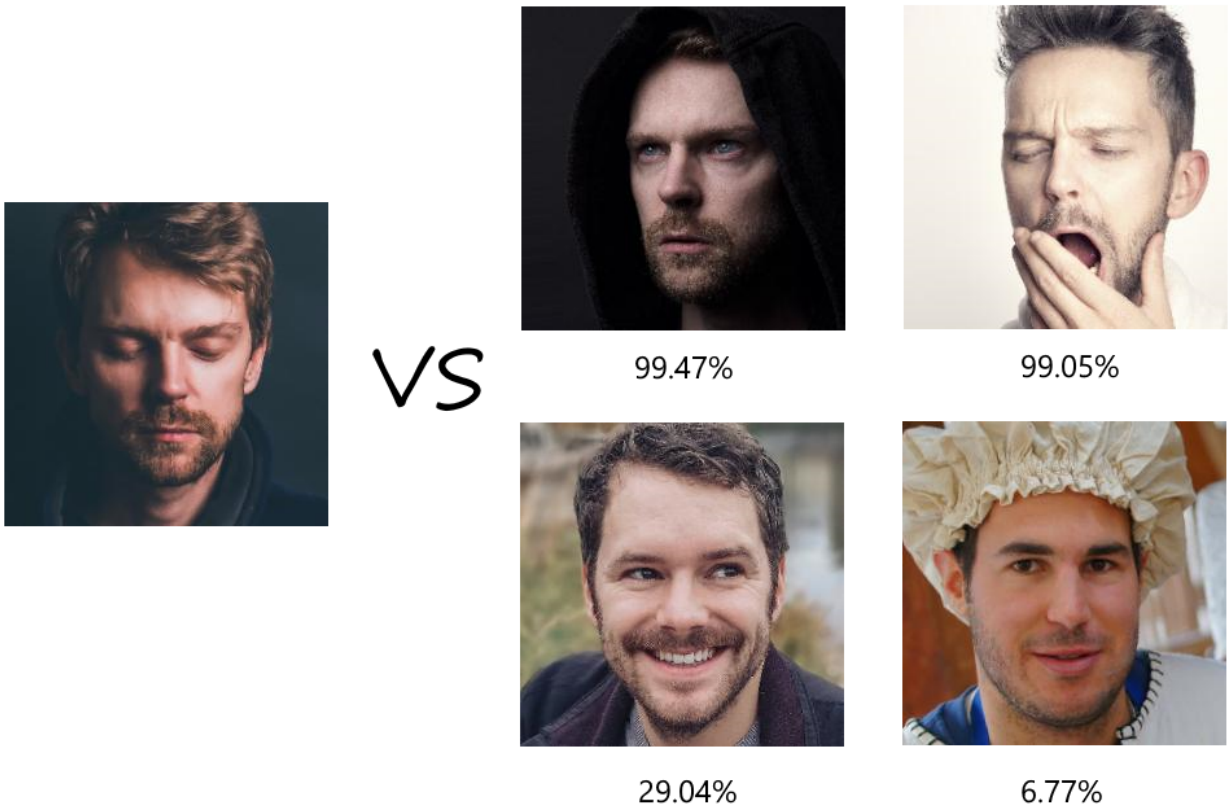


Figure 22: Matching

By means of *match* function defined by the *IDescriptorMatcher* interface it is possible to match a pair of descriptors with each other or a single descriptor with a descriptor batch (see section "Descriptor batch" for details on batches).

A simple rule to help you decide which storage to opt for:

- when searching among less than a hundred descriptors use separate *IDescriptor* objects;
- when searching among bigger number of descriptors use a batch.

When working with big data, a common practice is to organize descriptors in several batches keeping a batch per worker thread for processing.

Be aware that descriptor matching is not thread-safe, so you have to create a matcher object per a worker thread.

7.2.5 Descriptor Indexing

7.2.5.1 Using HNSW

To accelerate a descriptor matching process, you can create a **special index** for a descriptor batch. With the index, matching becomes a two-stage process:

First stage: build an indexed data structure — index — by using `IIndexBuilder`. This is quite a slow process, so it is not supposed to be done frequently. To build it, you can:

```
  - Append the `IDescriptor` or `IDescriptorBatch` objects
  - Use the `IIndexBuilder::buildIndex` build method
```

Second stage: use the index to quickly search the nearest neighbors for passed descriptors.

There are two types of indexes:

- `IDenseIndex`
  Read-only. Loading faster than `IDynamicIndex`. Once loaded, there are no performance differences in terms of searching between the two indexes.
- `IDynamicIndex`
  Editable. Allows you to append and remove descriptors. If you remove descriptors, they are removed from the graph for searching.
  To save `IDynamicIndex` with removed descriptors, first, call `eraseRemovedDescriptors` from `IDynamicIndex` structure. The state of the stored dynamic search index is not guaranteed for implementation reasons. If the descriptors are successfully erased, the remaining ID will move up. The shift depends on the number of removed handles. If the index state after erasing is valid, you can continue to use it for searching, otherwise you will have to rebuild it. > **Important:** We recommend to avoid operations that remove descriptors and rebuild the index by calling `IIndexBuilder::buildIndex` from a new set of descriptors and save the result as the dynamic index one more time.

You can only build a dynamic index. To get a dense index, you need to make it via deserialization. If you have several processes that might need to search in the index, do one of the following:

- Build an index for every process separately.
  > **Warning:** Building an index is a slow process.
- Build an index once and serialize it to a file.

7.2.5.2 Index serialization

To serialize an index, use the `IDynamicIndex::saveToDenseIndex` or `IDynamicIndex::saveToDynamicIndex` methods.

To deserialize an index, use the `IFaceEngine::loadDenseIndex` or `IFaceEngine::loadDynamicIndex` methods.

Important notes:

- Index files are not cross-platform. If you serialize an index on some platform, it is only usable on that exact platform. An operating system, as well as a different CPU architecture, may break compatibility.

- Embedded and 32-bit desktop platforms do not support the HNSW index.
- After large index files are loaded into RAM, the first lookup may take additional time due to process allocations. We recommend that you perform an idle search of descriptors to warm up.

7.2.5.3 Dynamic index evaluation scheme. This feature is experimental. Backward compatibility is not guaranteed.

In LUNA SDK v.5.17.0 and later, you can remove descriptors from a dynamic index in amounts of up to 80-90% of the total count. Deleting descriptors affects the internal structure of the index. The number of removed descriptors increases. For this reason, you must assess an index state.

7.2.5.3.1 Simple rules

- Call `isValidForSearch` every 10% of deletions from the original number of descriptors.
- Call `evaluate` after removing of 60% descriptors and every 10% of deletions after.
- Rebuilding an index is mandatory in a case of getting `DIS_INVALID`.
- Rebuilding an index is recommended if your index coefficient values are less than the ones in the table below (`searchForEvaluation` = 20):

Index size	Value
10M	0.5
20M	0.4
30M	0.4
40M	0.35

7.2.5.3.2 isValidForSearch method Call the `isValidForSearch` method after every removal of 10% of the original descriptor count. This method returns an index state. If the received state differs from `DIS_VALID`, you must rebuild the index to avoid unpredictable behavior.

The method specification is presented below:

```
virtual ResultValue<FSDKError, DynamicIndexState> isValidForSearch() const
    noexcept = 0;
```

Where available range of `DynamicIndexState` is:

```
    enum DynamicIndexState : uint8_t {
        DIS_INVALID = 0, //!< DIS_INVALID   - index is invalid for search.
```

```
            DIS_VALID,          //!< DIS_VALID     - index is valid for search.
            DIS_UNKNOWN,        //!< DIS_UNKNOWN   - index state is unknown.
            DIS_COUNT
        };
```

7.2.5.3.3 evaluate method Call the `evaluate` method after removing 60% of the original descriptor count.

The `evaluate` method takes significantly longer to run compared to `isValidForSearch`. You can specify `searchForEvaluation` and `numThreads` in the `IndexBuilder::Settings` section in faceengine.conf to tune it. The number of threads `numThreads` should be selected not greater than the number of cores in the system and not less than 0. By default, the number of threads is 0 and corresponds to the number of available cores.

The larger the `searchForEvaluation` value is, the more precise the evaluation will be, and the longer `evaluate()` method will run.

The method specification is presented below:

```
virtual ResultValue<FSDKError, float> evaluate() const noexcept = 0;
```

The method returns the status and the numerical value. The score is in the range [0, 1]

The table below shows estimated execution time, in minutes:

> `searchForEvaluation` is LengthSearch.

Index size	LengthSearch 20	LengthSearch 50	LengthSearch 100	LengthSearch 200
1.6M	1.65	2.44	2.73	3.19
10M	5.60	8.61	16.56	28.43
30M	22.10	32.03	39.60	58.63

```
Processor: Intel Xeon Skylake (IBRS)
Number of CPU cores: 32
CPU clock speed: 2.1 GHz
RAM capacity: 113 GB
```

It is necessary to rebuild the index after receiving the `DIS_INVALID` state regardless of the `value`. We recommend you to rebuild the index in the `DIS_VALID` state when the `value` is below the threshold.

If the index state is `DIS_INVALID`, you can save it to a file and load subsequently. The following method can be used to get a descriptor using its identifier:

```
virtual Result<FSDKError> descriptorByIndex(const DescriptorId index,
    IDescriptor* descriptor) const noexcept = 0;
```

8 System Requirements

8.1 Windows OS installations

We support 64-bit versions of the following operating systems:

Desktop/workstation environment:

- Windows 10 version 1909 or newer is required. Older versions are not supported.

Server environment:

- Windows Server 2016 or newer is required. Older versions are not supported.

Supported compiler:

- Visual Studio 17 2022. Other compilers may work but were not tested.

Note 1: FaceEngine requires a 64-bit version of Visual C++ Redistributable for Visual Studio 2022 to operate. The redistributable installer may be obtained from Microsoft via this link:

https://learn.microsoft.com/en-us/cpp/windows/latest-supported-vc-redist

8.2 Linux OS installations

We support the following operating systems:

- CentOS 8.2 64-bit
- AlmaLinux 8
- Ubuntu 2404 LTS 64-bit.

Supported compiler for CentOS 8.2 64-bit:

- GCC = GNU 7.5.0

Supported compiler for Ubuntu 2404 LTS 64-bit:

- GCC = GNU 13.2.0

Other compilers may work but were not tested.

Important notes:

- 32-bit OS on x86_64 CPU are not supported.
- Your OS should run glibc version 2.17 (CentOS) or 2.39 (Ubuntu), or newer.
- System locale must be US English. Specifically LC_NUMERIC=en_US.UTF-8.

9 Hardware requirements

9.1 Server / PC installations

See “Appendix A. Specifications” for information about hardware used for performance measurements.

9.1.1 General considerations

Be warned, that not all algorithms in the SDK have GPU or NPU implementations. If the desired algorithm doesn’t have a GPU or NPU implementation, a fallback to the CPU implementation has to be made. In this case, one should take care of possible memory transfers and latency they cause. Please see the algorithm implementation matrix for details.

Neural network	CPU	CPU AVX2	NPU Atlas	GPU
FaceDet_v2_<detector_type>_first_<device>.plan	yes			yes
FaceDet_v2_<detector_type>_second_<device>.plan	yes	yes		yes
FaceDet_v2_<detector_type>_third_<device>.plan	yes	yes		yes
FaceDet_v3_<version>_<device>.plan	yes	yes	yes	yes
FaceDet_v3_redetect_<version>_<device>.plan	yes	yes		yes
model_subjective_quality_<version>_<device>.plan	yes	yes		yes
headpose_v4_<device>.plan	yes	yes	yes	yes
ags_v3_<device>.plan	yes	yes	yes	yes
attributes_estimation_<device>.plan	yes	yes		yes
portrait_style_<version>_<device>.plan	yes	yes		yes
background_<version>_<device>.plan	yes	yes		yes
emotion_recognition_<version>_<device>.plan	yes	yes		yes
glasses_estimation_v2_<device>.plan	yes	yes		yes
eyes_estimation_flwr8_<device>.plan	yes	yes		yes
eye_status_estimation_<device>.plan	yes	yes		yes
eyes_estimation_ir_<device>.plan	yes	yes		yes
gaze_<version>_<device>.plan	yes	yes		yes
red_eye_<version>_<device>.plan	yes	yes		yes
gaze_ir_<version>_<device>.plan	yes	yes		yes
overlap_estimation_v1_<device>.plan	yes	yes		yes

Neural network	CPU	CPU AVX2	NPU Atlas	GPU
mouth_estimation_<version>_<device>.plan	yes	yes		yes
face_occlusion_v1_<device>.plan	yes	yes		yes
mask_clf_<version>_<device>.plan	yes	yes		yes
ppe_estimation_<version>_<device>.plan	yes	yes		yes
orientation_<device>.plan	yes	yes		yes
LNet_precise_<version>_<device>.plan	yes	yes		yes
LNet_ir_precise_<version>_<device>.plan	yes	yes		yes
slnet_<version>_<device>.plan	yes	yes		yes
liveness_model_<version>_<device>.plan	yes	yes		yes
depth_estimation_<device>.plan	yes	yes		yes
eyebrow_estimation_<version>_<device>.plan	yes	yes		yes
flying_faces_liveness_<version>_<device>.plan	yes	yes		yes
rgbm_liveness_<device>.plan	yes	yes		yes
rgbm_liveness_pp_hand_frg_<device>.plan	yes	yes		yes
natural_light_<device>.plan	yes	yes		yes
head_wear_<version>_<device>.plan	yes	yes		yes
fisheye_<version>_<device>.plan	yes	yes		yes
human_<version>_<device>.plan	yes	yes		yes
human_redetect_<device>.plan	yes	yes		yes
human_attributes_<version>_<device>.plan	yes	yes		yes
reid_<reid_type>_102_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_103_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_104_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_105_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_106_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_107_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_108_<device>.plan	yes	yes		yes
reid_<reid_type>_109_<device>.plan (deprecated)	yes	yes		yes
reid_<reid_type>_110_<device>.plan (deprecated)	yes	yes		yes

Neural network	CPU	CPU AVX2	NPU Atlas	GPU
reid_<reid_type>_112_<device>.plan	yes	yes		yes
reid_<reid_type>_113_<device>.plan	yes	yes		yes
cnn56b_<device>.plan	yes	yes		yes
cnn56m_<device>.plan	yes	yes		yes
cnn57b_<device>.plan	yes	yes	yes	yes
cnn58b_<device>.plan	yes	yes		yes
cnn59m_<device>.plan	yes	yes	yes	yes
cnn62b_<device>.plan	yes	yes		yes
cnn65b_<device>.plan	yes	yes		yes
oneshot_rgb_liveness_<version>_model_1_<device>.plan	yes	yes		yes
oneshot_rgb_liveness_<version>_model_2_<device>.plan	yes	yes		yes
oneshot_rgb_liveness_<version>_model_3_<device>.plan	yes	yes		yes
oneshot_rgb_liveness_<version>_model_7_<device>.plan	yes	yes		yes
crowd_<version>_<device>.plan	yes	yes		yes
depth_liveness_v2_<device>.plan	yes	yes	yes	yes
depth_rgb_<version>_<model_id>_<device>.plan	yes	yes	yes	yes
vlTracker_detection_<device>.plan	yes	yes	yes	yes
vlTracker_template_<device>.plan	yes	yes	yes	yes
vlTracker_update_<device>.plan	yes	yes	yes	yes

9.1.2 CPU requirements

For NN with "*_cpu.plan" in names, CPU should support at least the SSE4.2 instruction set.

For NN with "*_cpu-avx2.plan" in names, AVX2 instruction set support is required for the best performance.

> Only 64-bit CPUs are supported.

If in doubt, consider checking your CPU specifications at the following websites:

- Intel CPU: http://ark.intel.com
- AMD CPU: http://products.amd.com.

9.1.3 GPU requirements

For GPU acceleration an NVIDIA GPU is required. The following architectures are supported:

- Pascal or newer
- Compute Capability. The version depends on the platform (see "Requirements for GPU acceleration")

A minimum of 6GB or dedicated video RAM is required. 8 GB or more VRAM recommended.

9.1.4 The number of SDK threads while using GPU

If `Runtime::numThreads` in `runtime.conf` is not -1, then the SDK creates `numThreads - 1` worker threads. Besides, the user-created thread that initializes the SDK is used as a worker thread too.

In addition to worker threads, the SDK creates approximately `GPU_count + 1` CUDA threads, where `GPU_count` is the number of GPUs in the system. The number of CUDA threads may slightly vary depending on drivers.

Example Assume `Runtime::numThreads=4` and there is 1 GPU. If we list all threads in the process, we get something like the following:

```
thread id=4162 (unittest_core_t)
thread id=4170 (Worker 1)
thread id=4171 (Worker 2)
thread id=4172 (Worker 3)
thread id=4186 (cuda00001400006)
thread id=4273 (cuda-EvtHandlr)
```

Here the thread `unittest_core_t` is created by the user ("main thread") and is used to initialize the SDK. It will be used as one of worker threads by the SDK. `Worker 1`, `Worker 2`, `Worker 3` are 3 workers created by the SDK. The 2 CUDA threads are also created by the SDK (1 GPU + 1).

9.1.5 NPU requirements

Huawei Atlas NPU was utilized with the following drivers and additional SW installed:

Drivers:

- Version = 20.2.0
- ascendhal_version = 4.0.0
- aicpu_version = 1.0
- tdt_version = 1.0
- log_version = 1.0
- prof_version = 2.0

- dvppkernels_version = 1.1
- tsfw_version = 1.0
- required_firmware_firmware_version = 1.0

Firmware:

- Version = 1.76.22.3.220
- firmware_version = 1.0

Toolkit:

- Version = 1.76.22.3.220

9.1.6 RAM requirements

System memory consumption differs depending on a usage scenario and is proportional to the number of worker threads. This is true for both CPU (think system RAM) and GPU (think VRAM) execution modes.

For example, in CPU execution mode 1GB RAM is enough for a typical pipeline, which consists of a face detector and a face descriptor extractor running on a single core (one worker thread) and processing 1080p input images with 10-12 faces on average. If this setup is scaled up to 8 worker threads, overall memory consumption grows up to 8GB.

It is recommended to assume at least 1GB of free RAM per worker thread.

9.1.7 Storage requirements

FaceEngine requires 1GB of free space to install. This includes model data for both CPU and GPU execution modes that should be redistributed with your application. If only one execution mode is planned, reduce space requirements by half.

9.1.8 Approaches to software design targeting different hardware

When performing inference on different hardware, several key differences should be taken into account to reach maximum possible performance:

9.1.8.1 CPU

Key points:

- Memory used by the inference engine is physically located on the same chips where OS and business logic data reside. Source data (images/video frames) also reside there.
- The CPU is general-purpose hardware, not tailored for many operations specific to NN inference.

Implications:

- No memory transfers ever performed, memory access latency is low. the CPU is easily saturated with work.
- Both memory and CPU may receive additional pressure from background processes.

Recommendations:

- Don't expect profit from batching. If the software isn't expected to ever run/support GPU or NPU, don't implement it at all. Instead, consider culling computation-heavy algorithms early (e.g. check head pose and AGS score before attempting to extract a descriptor in order to avoid the extraction for bad faces).
- Use tools like taskset() to isolate different types of workload on process level on servers.
- Consider running a separate SDK process per node on NUMA systems. Note, that SDK itself is not NUMA-aware.

9.1.8.2 GPU/NPU

Key points:

- Memory used by the inference engine is physically located on the device and source data (images/video frames) is on the host memory.
- While servers typically use DDR memory, GPU/NPU devices prefer GDDR, which offers higher throughput at the cost of higher latency.
- GPU/NPU devices process excessive amounts of data in hundreds/thousands of threads without external interference. In addition, they implement specialized instructions for many typical NN inference operations.
- GPU/NPU are fed with work by the CPU.

Implications:

- Memory transfers should be taken into account. Such transfers typically take place by means of the PCI-e bus and the bus may become the performance bottleneck. GPU/NPU generally needs much more input data to saturate it with work.

Recommendations:

- Batch multiple source images together and do inference for the entire match at once. This helps to saturate both the bus and the device. See recommended batch sizes in chapter Appendix A. Specifications.

- Take care of memory residence. While SDK will do an implicit memory transfer for you, in some cases it is beneficial to do this yourself. E. g. Both Tesla and Atlas cards implement on-board hardware accelerated decoders for JPEG and h264 formats. If your software utilizes these decoders, don't transfer the decoder output to the host memory. Instead, pass the device pointer to the SDK directly. Note, that SDFK Image class can wrap an existing memory pointer at no cost.

- Take care of device work scheduling. The general rule of thumb:

- Don’t acces the same device from multiple threads/processes, this may involve kernel level locks or be unsupported at all
- Access different devices from different threads/processes. This way work scheduling is less likely to be CPU-bound.
- Workload isolation recommendations for the CPU also apply here.

SDK algorithms are device-bound. To support multiple devices in one process, you are required to create each algorithm implementation you need on a per-device basis and bind it to the corresponding device as shown in the example below:

```
int32_t npuDeviceIndex = 1;
fsdk::LaunchOptions launchOptions;
launchOptions.deviceClass = fsdk::DeviceClass::NPU_ASCEND;
launchOptions.npuDevice = npuDeviceIndex;

auto result = faceEngine->createDetector(
      detectorType,
      fsdk::SensorType::Visible,
      &launchOptions
    );
ASSERT_TRUE(result.isOk());

auto detector = result.getValue();
```

GPU specific recommendations

GPUs tend to be harder to saturate with work. Consider bigger batches.

NPU specific recommendations

Atlas 300I NPU is designed such that there are 4 different NPU devices per accelerator card. This means that you have to design your software for multi-device scenarios from the ground up to achieve the best performance. The card has a PCI-e x8 bus connector and each NPU device consumes x2 lanes from it; the bus is likely to become the bottleneck. Atlas 300I NPU is saturated with work quite easily; batching makes sense for some particularly lightweight NNs mostly. Memory operations on the device (copy, clears) are particularly slow.

9.1.9 Requirements for GPU acceleration

Recommended versions of CUDA:

- For Win64 - CUDA Toolkit 11.6
- For Linux(CentOS, AlmaLinux) - CUDA Toolkit 11.4
- For Linux(Ubuntu) - CUDA Toolkit 12.4

The most current version of these release notes can be found online at http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html.

> **Important:** For Win64 and Linux (CentOS, AlmaLinux) there are additional requirements - Compute Capability 6.1 or higher.

CUDA version on Linux can be found using command below:

```
$nvidia-smi
```

CUDA version on Windows can be found in `Control Panel\Programs\Programs and Features` as in figure below

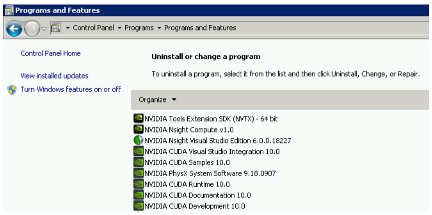


Figure 23: CUDA version on Win

> We recommend that you use suggested version of CUDA for your operating system. But if your version is older than required, we do not guarantee, that it will work successfully. For details about CUDA Compatibility, see https://docs.nvidia.com/deploy/cuda- compatibility/index.html.

9.2 Embedded installations

9.2.1 CPU requirements

Supported CPU architectures:

- ARMv7-A;
- ARMv8-A (ARM64).

10 Migration guide

10.1 Overview

Here you can find information about important changes in the next releases of LUNA SDK.

10.2 v.5.32.0

10.2.1 IQualityEstimator

Since version 5.32.0 the `IQualityEstimator` has been deprecated. Use `ISubjectiveQualityEstimator` instead.

Example of code (before version 5.32.0):

```
// Create quality estimator
fsdk::ResultValue<fsdk::FSDKError, fsdk::IQualityEstimatorPtr>
    resQualityEstimator =
     faceEngine->createQualityEstimator();
if(!resQualityEstimator) {
     std::cerr << "Failed to create quality estimator. Reason: " <<
         resQualityEstimator.what() << std::endl;
     return -1;
}
fsdk::IQualityEstimatorPtr qualityEstimator = resQualityEstimator.
    getValue();

// Estimate using SubjectiveQuality (v2)
fsdk::SubjectiveQuality subjectiveQuality;
fsdk::Result<fsdk::FSDKError> result = qualityEstimator->estimate(warp,
    subjectiveQuality);
```

Example of code (from version 5.32.0):

```
// Create subjective quality estimator
fsdk::ResultValue<fsdk::FSDKError, fsdk::ISubjectiveQualityEstimatorPtr>
     resSubjectiveQualityEstimator =
     faceEngine->createSubjectiveQualityEstimator();
if(!resSubjectiveQualityEstimator) {
     std::cerr << "Failed to create subjective quality estimator. Reason:
          " << resSubjectiveQualityEstimator.what() << std::endl;
     return -1;
}
```

```
        fsdk::ISubjectiveQualityEstimatorPtr subjectiveQualityEstimator =
            resSubjectiveQualityEstimator.getValue();

        // Estimate using SubjectiveQuality
        fsdk::SubjectiveQuality subjectiveQuality;
        fsdk::Result<fsdk::FSDKError> result = subjectiveQualityEstimator->
            estimate(warp, subjectiveQuality);
```

10.2.2 ILicense::loadFromFile(const char* path)

From v.5.32.0 the method `loadFromFile(const char* path)` (See `ILicense.h`) has been removed. Please use the method `loadFromFile(const char* path, const fsdk::ISettingsProvider* settings)` instead.

10.3 v.5.31.0

10.3.1 IDeepFakeEstimator

Since version 5.31.0 the IDeepFakeEstimator has only one working mode: - the enum `fsdk::DeepFakeMode` has been removed - the "defaultEstimatorType" parameter in the section "DeepFakeEstimator::Settings" of the configuration file faceengine.conf has also been removed. - the create-function `fsdk::IFaceEngine::createDeepFakeEstimator` takes only one argument - the fsdk::LaunchOptions pointer.

Example of code (before version 5.31.0):

```
        fsdk::LaunchOptions launchOptions = {};
        fsdk::experimental::DeepFakeMode mode = fsdk::experimental::DeepFakeMode
            ::M2;

        fsdk::ResultValue<fsdk::FSDKError, fsdk::experimental::
            IDeepFakeEstimatorPtr> resultValue =
             faceEngine->createDeepFakeEstimator(mode, &launchOptions);
        // ...
```

Example of code (from version 5.31.0):

```
        fsdk::LaunchOptions launchOptions = {};
        fsdk::ResultValue<fsdk::FSDKError, fsdk::experimental::
            IDeepFakeEstimatorPtr> resultValue =
             faceEngine->createDeepFakeEstimator(&launchOptions);
        // ...
```

10.4 v.5.27.0

10.4.1 Multithreading usage of factory functions (for example: createAGSEstimator, createHeadWearEstimator...)

Before 5.27.0 we describe rules of FaceEngine interface usage in matter of multithreading in Chapter 1 Core Concepts. Now, we assert this rules by adding strict checks by thread_id. Now if rules is not followed user will get errors.

10.5 v.5.24.0

10.5.1 IDetector

The `FaceDetV1` has been removed since v.5.24.0.

10.6 v.5.23.0

10.6.1 IImageTransfer

Since version 5.23.0, a method for a single image in the IImageTransfer interface has been removed.

Example of code (before version 5.23.0):

```
auto result = imageTransfer->transfer(image, fsdk::Image::
    MemoryResidence::MemoryGPU);
// or
auto result = imageTransfer->transfer(images, fsdk::Image::
    MemoryResidence::MemoryGPU);
```

Example of code (from version 5.23.0):

```
auto result = imageTransfer->transfer(images, fsdk::Image::
    MemoryResidence::MemoryGPU);
```

10.6.2 IDetector

The `FaceDetV1` has been deprecated since v.5.23.0. Use `FaceDetV3` instead.

10.7 v.5.22.0

10.7.1 IHeadPoseEstimator

Since version v.5.22.0, an estimation method of `IHeadPoseEstimator`, based on `Landmarks68` has been dropped. Accordingly, the configuration block - `"HeadPoseEstimator::Settings"`, which allows the user to define which method to use, has also been dropped.

10.7.2 IHeadPoseEstimator and IAGSEstimator

Since version v.5.22.0, `IHeadPoseEstimator` and `IAGSEstimator` have been reconsidered and reinstated.

10.8 v.5.20.0

10.8.1 ILivenessFlowEstimator

Since v.5.20.0 the `ILivenessFlowEstimator` estimator has been removed.

10.9 v.5.19.0

10.9.1 ILivenessFlowEstimator

Since v.5.19.0 the `ILivenessFlowEstimator` estimator has been deprecated. If you still need this estimator, please, contact VisionLabs for details.

10.10 v.5.18.0

10.10.1 IChildEstimator

Since v.5.18.0 the `IChildEstimator` estimator has been removed. Use the `IAttributeEstimator` (See `IAttributeEstimator.h`) with `IAttributeEstimator::EstimationRequest::estimateAge` instead.

10.10.2 IHeadAndShouldersLivenessEstimator

Since v.5.18.0 the `IHeadAndShouldersLivenessEstimator` estimator has been removed. If you still need this estimator, please, contact VisionLabs for details.

10.11 v.5.17.0

10.11.1 IHeadAndShouldersLivenessEstimator

Since v.5.17.0 the estimator `IHeadAndShouldersLivenessEstimator` has been deprecated (See `IHeadAndShouldersLivenessEstimator.h`). If you need this estimator, please, contact VisionLabs for details.

10.11.2 IChildEstimator

Since v.5.17.0 the estimator `IChildEstimator` has been deprecated (See `IChildEstimator.h`). Use the `IAttributeEstimator` (See `IAttributeEstimator.h`) with `IAttributeEstimator::EstimationRequest::estimateAge` instead.

Example of code (before version v.5.17.0):

```
        // Create child estimator.
        auto resChildEstimator = faceEngine->createChildEstimator();
        if(!resChildEstimator) {
            std::cerr << "Failed to create child estimator instance. Reason: "
                << resChildEstimator.what();
            std::cerr << std::endl;
            return -1;
        }
        fsdk::IChildEstimatorPtr childEstimator = resChildEstimator.getValue();

        // Get child estimation.
        fsdk::ChildEstimation childEstimation;
        fsdk::Result<fsdk::FSDKError> childEstimationResult = childEstimator->
            estimate(warp, childEstimation);
        if(childEstimationResult.isOk()) {
            std::cout << "\nChild estimate:";
            std::cout << "\nchildScore: " << childEstimation.childScore << " (
                range [0, 1])";
            std::cout << "\nis child: " << childEstimation.isChild << " (1 - is
                child, 0 - is adult)";
            std::cout << std::endl;
        } else {
            std::cerr << "Failed child estimation. Reason: " <<
                childEstimationResult.what() << std::endl;
        }
```

Example of code (from version v.5.17.0):

```
        // Create attribute estimator.
        auto resAttributeEstimator = faceEngine->createAttributeEstimator();
        if(!resAttributeEstimator) {
            std::cerr << "Failed to create attribute estimator instance. Reason:
                " << resAttributeEstimator.what();
            std::cerr << std::endl;
            return -1;
        }
        fsdk::IAttributeEstimatorPtr attributeEstimator = resAttributeEstimator.
            getValue();

        // Get attribute estimation.
        using AttrsRequest = fsdk::IAttributeEstimator::EstimationRequest;
        AttrsRequest attributesRequest = AttrsRequest::estimateAge;
        fsdk::IAttributeEstimator::EstimationResult attributeEstimation;

        fsdk::Result<fsdk::FSDKError> attributeEstimatorResult =
            attributeEstimator->estimate(warp, attributesRequest,
                attributeEstimation);

        if(attributeEstimatorResult.isOk()) {
            std::cout << "\nAttribute estimate:";
            std::cout << "\nage: " << attributeEstimation.age.value() << " (in
                years)" << std::endl;
            std::cout << std::endl;
        } else {
            std::cerr << "Failed to make attribute estimation. Reason: " <<
                attributeEstimatorResult.what();
            std::cerr << std::endl;
        }
```

10.11.3 Index

Since v.5.17.0 IDynamicIndex can be saved as a file to hard disc after removing of descriptors.

Example of code (before version v.5.17.0):

```
    // index with removed descriptors could not be saved
    for (std::size_t i = 0; i < nRemoved; ++i) {
        auto resRemove = index->removeDescriptor(i);
        if (resRemove.isError()) {
            // process error
```

```
        ...
    }
}
```

Example of code (from version v.5.17.0):

```
// remove descriptors
for (std::size_t i = 0; i < nRemoved; ++i) {
    auto resRemove = index->removeDescriptor(i);
    if (resRemove.isError()) {
        // process error
        ...
    }
}

// erase descriptors
auto resEraseRemovedDescs = index->eraseRemovedDescriptors();
if(resEraseRemovedDescs.isError()) {
    // process error
    ...
}

// get map of new descriptors
auto map = resEraseRemovedDescs.getValue();
for (std::size_t i = 0; i < count; ++i) {
    // if the old ID is not found, the error InvalidDescriptorId will be
       returned
    auto resMapFind = map->getId(i);
        if (resMapFind.isError()) {
            // process error or skip not found id
            ...
        }
        // we can get new id by old id
        auto newId = resMapFind.getValue();

}

// now we can save index
auto resSave = index->saveToDynamicIndex("your_index_name.bin");
if (resSave.isError()) {
    // process error
}
```

10.11.4 FishEyeEstimator

Since v.5.13.0 method `estimate` of `IFishEyeEstimator` by crop and detection has been deprecated (See `IFishEyeEstimator.h`). Use `estimate` by warped image instead.

Example of code (before version 5.13.0):

```
fsdk::FishEyeEstimation estimation;
fsdk::Result<fsdk::FSDKError> res = fishEyeEstimator->estimate(image,
   detection, estimation);
```

Example of code (from version 5.13.0):

```
fsdk::FishEyeEstimation estimation;
fsdk::Result<fsdk::FSDKError> res = fishEyeEstimator->estimate(warp,
   estimation);
```

10.12 v.5.6.0

10.12.1 Vector2

Since v.5.6.0, the member `array` in `fsdk::Vector2` has been removed. You should use the `x` andy members instead of the removed `array` one.

Example of code (before version 5.6.0):

```
fsdk::Vector2<int> vector2;
vector2.x = 10;
vector2.y = 20;
// or
vector2.array[0] = 10;
vector2.array[1] = 20;
```

Example of code (from version 5.6.0):

```
fsdk::Vector2<int> vector2;
vector2.x = 10;
vector2.y = 20;
```

10.12.2 BlackWhiteEstimator

Since v.5.6.0 method `estimate` of `IBlackWhiteEstimator` by full image has been deprecated (See `IBlackWhiteEstimator.h`). Use `estimate` by warped image instead.

Example of code (before version 5.6.0):

```
bool isGray = false;
Result<FSDKError> res = BlackWhiteEstimator->estimate(fullImage, isGray)
    ;
```

Example of code (from version 5.6.0):

```
fsdk::ImageColorEstimation estimation;
Result<FSDKError> res = BlackWhiteEstimator->estimate(warp, estimation);
```

10.13 v.5.5.0

From v.5.5.0 the default value of `numThreads` (runtime.conf) was replaced by `-1`. Which means that will be taken the maximum number of available threads. This number of threads is equal to according number of available processor cores.

Example of setting (before version 5.5.0):

```
<param name="numThreads" type="Value::Int1" x="4" />
```

Example of setting (from version 5.5.0):

```
<param name="numThreads" type="Value::Int1" x="-1" />
```

From v.5.5.0 the method `loadFromFile(const char* path)` (See `ILicense.h`) is deprecated. The use is allowed, but can be useless. Please use the method `loadFromFile(const char* path, const fsdk::ISettingsProvider* settings)` instead.

10.13.0.1 Examples of code

Example of code (before version 5.5.0):

```
const bool isLicenseFileLoadedSuccessfully = license->loadFromFile(path)
    .isOk());
```

Example of code (from version 5.5.0):

```
auto resSettings = fsdk::createSettingsProvider("License Config Path");
if (!resSettings.isOk()) {
    return -1;
}

fsdk::ISettingsProviderPtr settings = resSettings.getValue();

// Create new license from file
const bool isLicenseFileLoadedSuccessfully = license->loadFromFile(path,
     settings).isOk());
```

10.14 v.5.2.0

From v.5.2.0 the 101 version of human descriptor is not supported, it was changed by 104. Currently, three versions are available: 102 (tracker), 103 (precise), 104 (regular). It means that all instances (such as `IDescriptorExtractor`, `IDescriptorMatcher` and etc.) cannot be created with the version 101.

10.15 v.5.1.0

From version v.5.1.0 `IHeadPoseEstimatorPtr` and `IAGSEstimatorPtr` are deprecated. Use `IBestShotQualityEstimatorPtr` instead.

Note. AGS score thresholds are different for `IAGSEstimatorPtr` and `IBestShotQualityEstimatorPtr`. Read more on the BestShotQuality estimation page.

10.16 v.5.0.0

10.16.1 Objects creation

The `fsdk::acquire(...)` method for the pointer acquiring for `IFaceEngine` objects is not allowed for usage starting from version 5.0.0. In addition, the types of values returned from the `create` methods of `IFaceEngine` were changed.

Most of the `create` methods now return the following structure - `fsdk::ResultValue<fsdk::FSDKError, ObjectClassPtr>` Thus it is easy to check the correctness of the result (using one of the following methods `result.isOk()` or `result.isError()`) and get an error (using the `result.getError()` method). The `result.what()` method can be used to get the text description of the error.

10.16.1.1 Examples of code

Example of code (before version 5.0.0):

```
fsdk::IAttributeEstimatorPtr estimator = fsdk::acquire(faceEngine->
    createAttributeEstimator());
if (estimator.isNull())  {
    std::cout << "Object pointer is nullptr" << std::endl;
    ... // process error
}
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::IAttributeEstimatorPtr>
    resEstimator = faceEngine->createAttributeEstimator();
if (resEstimator.isError())  {
    std::cout << "Error: " << resEstimator.what() << std::endl;
    ... // process error
}

fsdk::IAttributeEstimatorPtr estimator = resEstimator.getValue();
```

10.16.2 Interface of ILicense

From version v.5.0.0 we changed the interface of `ILicense`. Now all methods of this class return `fsdk::Result<fsdk::FSDKError>`, `fsdk::ResultValue<fsdk::FSDKError, bool>` or `fsdk::ResultValue<fsdk::FSDKError, uint32_t>` instead of `bool`.

10.16.2.1 Examples of code

Example of code (before version 5.0.0):

```
    const bool res = license->isActivated();
    if (!res) {
        /* error case code */
    }
```

Example of code (from version 5.0.0):

```
    const fsdk::ResultValue<fsdk::FSDKError, bool> result = license->
        isActivated();
    if (result.isError()) {
```

```
        /* error case code */
    }

    const bool value = result.getValue();
    if (!value) {
        /* false case code */
    }
```

From version v.5.0.0 we changed the arguments of methods `getExpirationDate` and `checkFeatureId` in class `ILicense`. Now the input arguments of `getExpirationDate` and `checkFeatureId` is `fsdk::LicenseFeature` instead of `uint32_t`. And the second argument of `getExpirationDate` was removed. The return value of `getExpirationDate` is `fsdk::ResultValue<fsdk::FSDKError, uint32_t>`.

Example of code (before version 5.0.0):

```
    long long expDate = 0;
    const bool result =
        license->getExpirationDate(static_cast<uint32_t>(fsdk::
           LicenseFeature::Detection), expDate);

    if (result == false) {
        /* error case code */
    }
```

Example of code (from version 5.0.0):

```
    const fsdk::ResultValue<fsdk::FSDKError, uint32_t> result =
        license->getExpirationDate(fsdk::LicenseFeature::Detection);

    if (result.isError()) {
        /* error case code */
    }

    const uint32_t expDate = result.getValue();
```

Example of code (before version 5.0.0):

```
    const bool res = license->checkFeatureId(static_cast<uint32_t>(fsdk::
       LicenseFeature::Detection));
    if (!res) {
        /* error case code */
    }
```

Example of code (from version 5.0.0):

```
const fsdk::ResultValue<fsdk::FSDKError, bool> result = license->
    checkFeatureId(fsdk::LicenseFeature::Detection);
if (result.isError()) {
    /* error case code */
}

const bool value = result.getValue();
if (!value) {
    /* false case code */
}
```

10.16.3 Interface of HumanLandmark

From version v.5.0.0 we changed the interface of `HumanLandmark`. Now member `point` doesn't store zero coordinates in the case when it is not visible. For this purposes we added member `visible` which stores `true` if point is visible.

Example of code (before version 5.0.0):

```
if (humanLandmark.point.x == 0 && humanLandmark.point.y == 0) {
    // point is not visible case code
}
else {
    // point is visible case code
}
```

Example of code (from version 5.0.0):

```
if (humanLandmark.visible == false) {
    // point is not visible case code
}
else {
    // point is visible case code
}
```

10.16.3.1 HumanDetectionType

Since v.5.19.0 the `HDT_POINTS` was dropped, but the the enum *`HumanDetectionType`* kept for backward compatibility

10.16.3.2 HumanLandmarks17

Since v.5.19.0 were dropped the `HumanLandmarks17`, special points for the body parts visible in the image, and the member function `getLandmarks17`, which was intended to return `HumanLandmarks17` Span.

10.16.3.3 IHumanLandmarksDetector

Since v.5.19.0 were dropped the `IHumanLandmarksDetector` - a human landmark(`HumanLandmarks17`) detector.

10.16.4 Interface of IDescriptorBatch

From version v.5.0.0 we renamed method `IDescriptorBatch::getDescriptorSize()` to `IDescriptorBatch::getDescriptorLength()`.

Example of code (before version 5.0.0):

```
uint32_t descriptorLength = descriptorBatch->getDescriptorSize();
```

Example of code (from version 5.0.0):

```
uint32_t descriptorLength = descriptorBatch->getDescriptorLength();
```

10.16.5 Interface of Detection

From version v.5.0.0 we changed the interface of the `Detection` structure. Now all members of this structure are private and could be available through the public methods.

Example of code (before version 5.0.0):

```
fsdk::Detection detection = ...; // Somehow initialized detection object
fsdk::Rect rect = detection.rect; // Get the detection rect
float score = detection.score; // Get the detection score
```

Example of code (from version 5.0.0):

```
fsdk::Detection detection = ...; // Somehow initialized detection object
fsdk::Rect rect = detection.getRect(); // Get the detection rect
float score = detection.getScore(); // Get the detection score
```

10.16.6 Interface of IDetector

From version v.5.0.0 we changed the interface of `IDetector` structure. Now method `detect` returns `ResultValue<FSDKError, Ref<IFaceDetectionBatch>>` instead of `ResultValue<FSDKError, Ref<IResultBatch<Face>>>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IResultBatch<fsdk::Face
    >>> detectorResult = faceDetector->detect(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Rect>(&imageRect, 1),
        detectionsCount,
        fsdk::DT_ALL);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IFaceDetectionBatch>>
    detectorResult = faceDetector->detect(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Rect>(&imageRect, 1),
        detectionsCount,
        fsdk::DT_ALL);
```

Also we changed input and output parameters of the method `redetectOne`. Now it takes `Image` and `Detection` instead of `Face`. And returns `ResultValue<FSDKError, Face>` instead of `ResultValue<FSDKError, bool>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, bool> redetectResult = faceDetector->
    redetectOne(face);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Face> redetectResult = faceDetector
    ->redetectOne(image, detection);
```

10.16.7 IFaceDetectionBatch

We added `IFaceDetectionBatch` structure to replace `IResultBatch<Face>`.

Example of code (before version 5.0.0):

```
fsdk::Ref<IResultBatch<Face>> resultBatch = ...; // Somehow get the
    IResultBatch<Face>
for (std::size_t i = 0; i < resultBatch->getSize(); ++i) {
    fsdk::Span<fsdk::Face> faces = resultBatch->getResults(i);
    for (auto& face : faces) {
        const fsdk::Rect& rect = face.detection.rect;
        const float score = face.detection.score;
        const fsdk::Landmarks5& lm5 = face.landmarks5.value();
        const fsdk::Landmarks68& lm68 = face.landmarks68.value();
        // Some code which uses received objects
    }
}
```

Example of code (from version 5.0.0):

```
fsdk::Ref<fsdk::IFaceDetectionBatch> faceDetectionBatch = ...; // Somehow
    get the IFaceDetectionBatch
for (std::size_t i = 0; i < faceDetectionBatch->getSize(); ++i) {
    fsdk::Span<const fsdk::Detection> detections = faceDetectionBatch->
        getDetections(i);
    fsdk::Span<const fsdk::Landmarks5> landmarks5 = faceDetectionBatch->
        getLandmarks5(i);
    fsdk::Span<const fsdk::Landmarks68> landmarks68 = faceDetectionBatch->
        getLandmarks68(i);
    for (std::size_t j = 0; j < detections.size(); ++j) {
        const fsdk::Rect& rect = detections[j].getRect();
        const float score = detections[j].getScore();
        const fsdk::Landmarks5& lm5 = landmarks5[j];
        const fsdk::Landmarks68& lm68 = landmarks68[j];
        // Some code which uses received objects
    }
}
```

10.16.8 Interface of IHumanDetector

From version v.5.0.0 we changed the interface of `IHumanDetector` structure. Now method `detect` returns `ResultValue<FSDKError, Ref<IHumanDetectionBatch>>` instead of `ResultValue<FSDKError, Ref<IResultBatch<Human>>>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IResultBatch<fsdk::Human
    >>> detectResult = humanDetector->detect(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Rect>(&imageRect, 1),
        detectionsCount,
        fsdk::DCT_ALL);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Ref<fsdk::IHumanDetectionBatch>>
    detectResult = humanDetector->detect(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Rect>(&imageRect, 1),
        detectionsCount,
        fsdk::HDT_ALL);
```

Also we changed input and output parameters of the method `redetectOne`. Now it takes `Image` and `Detection` instead of `Human`. And returns `ResultValue<FSDKError, Human>` instead of `ResultValue<FSDKError, bool>`.

Example of code (before version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, bool> redetectResult = humanDetector->
    redetectOne(human);
```

Example of code (from version 5.0.0):

```
fsdk::ResultValue<fsdk::FSDKError, fsdk::Human> redetectResult =
    humanDetector->redetectOne(image, detection);
```

10.16.9 IHumanDetectionBatch

Since v.5.19.0 were dropped the member function `getLandmarks17`, which was intended to return `HumanLandmarks17` Span.

We added `IHumanDetectionBatch` structure to replace `IResultBatch<Human>`.

Example of code (before version 5.0.0):

```
fsdk::Ref<IResultBatch<Human>> resultBatch = ...; // Somehow get the
    IResultBatch<Human>
for (std::size_t i = 0; i < resultBatch->getSize(); ++i) {
     fsdk::Span<fsdk::Human> humans = resultBatch->getResults(i);
```

```
    for (auto& human : humans) {
        const fsdk::Rect& rect = human.detection.rect;
        const float score = human.detection.score;
        const fsdk::Landmarks17& lm17 = face.landmarks5.value();
        // Some code which uses received objects
    }
}
```

Example of code (from version 5.0.0):

```
const fsdk::Ref<fsdk::IHumanDetectionBatch> humanDetectionBatch = ...; //
   Somehow get the IHumanDetectionBatch
for (std::size_t i = 0; i < humanDetectionBatch->getSize(); ++i) {
    fsdk::Span<const fsdk::Detection> detections = humanDetectionBatch->
       getDetections(i);
    fsdk::Span<const fsdk::HumanLandmarks17> landmarks = humanDetectionBatch
       ->getLandmarks17(i);
    for (std::size_t j = 0; j < detections.size(); ++j) {
        const fsdk::Rect rect = detections[j].getRect();
        const float score = detections[j].getScore();
        const fsdk::HumanLandmarks17 lm17 = landmarks[j];
        // Some code which uses received objects
    }
}
```

10.16.10 Interface of ILivenessFlyingFaces

From version v.5.0.0 we changed the interface of `ILivenessFlyingFaces` structure. Now both methods `estimate` take `Image` and `Detection` instead of `Face`.

Example of code (before version 5.0.0):

```
fsdk::LivenessFlyingFacesEstimation flyingFacesEstimation;
Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
   estimate(face, flyingFacesEstimation);
```

Example of code (from version 5.0.0):

```
fsdk::LivenessFlyingFacesEstimation flyingFacesEstimation;
Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
   estimate(
        image,
        detection,
```

```
            flyingFacesEstimation);
```

Example of code (before version 5.0.0):

```
Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
    estimate(
        fsdk::Span<const fsdk::Face>(&face, 1),
        fsdk::Span<fsdk::LivenessFlyingFacesEstimation>(&estimation, 1));
```

Example of code (from version 5.0.0):

```
fsdk::LivenessFlyingFacesEstimation flyingFacesEstimation;
Result<fsdk::FSDKError> flyingFacesResult = livenessFlyingFacesEstimator->
    estimate(
        fsdk::Span<const fsdk::Image>(&image, 1),
        fsdk::Span<const fsdk::Detection>(&detection, 1),
        fsdk::Span<fsdk::LivenessFlyingFacesEstimation>(&
            flyingFacesEstimation, 1));
```

10.17 v.3.10.1

10.17.1 Detector FaceDetV3 changes

From version 3.10.1 we changed the logic for image resizing in FaceDetV3 detector. Now you can change the minimum and maximum sizes of the faces that will be searched in the photo from the `faceengine.conf` configuration. To get new parameter which will be identical to old setting you need to set `minFaceSize`:

The old recommended `imageSize`=640 will be identical to new meaning of setting `minFaceSize`=20

```
config->setValue("FaceDetV3::Settings", "minFaceSize", 20);
```

and `imageSize`=320 will be identical to new meaning of setting `minFaceSize`=40

```
config->setValue("FaceDetV3::Settings", "minFaceSize", 40);
```

10.17.2 Detector FaceDetV1, FaceDetV2 changes

From version 3.10.1 we changed the name of parameter `minSize` to `minFaceSize` in `faceengine.conf` for FaceDetV1, FaceDetV2 detector types. The logic and default value for image resizing left unchanged.

11 Best practices

This section provides a set of recommendations and performance tips that you should follow to get optimal performance when running the LUNA SDK algorithms on your target device.

11.1 Thread pools

We recommend that you use thread pools for user-created threads when running LUNA SDK algorithms in a multithreaded environment. For each thread, LUNA SDK caches some amount of thread local objects under the hood in order to make its algorithms run faster next time the same thread is used at the cost of higher memory footprint. For this reason, we recommend that you reuse threads from a pool to avoid caching new internal objects and to reduce penalty of creating or destroying new user threads.

11.2 Estimator creation and inference

To optimize RAM usage and improve performance, create face engine objects once and reuse them whenever a new estimate is needed.

Recreating estimators repeatedly results in reopening their corresponding *.plan* files each time, which can be resource-intensive. These *.plan* files are cached individually upon loading and remain in memory until they are either flushed from the cache or the FaceEngine root object's destructor is called. By reusing existing objects, you avoid unnecessary overhead and ensure efficient resource management.

11.3 Using CPU and GPU models for network inference

To ensure optimal performance and accuracy when using LUNA SDK, it is essential to follow our recommendations for CPUs and GPUs based on the type of workload and network configurations.

11.3.1 CPU recommendations

- **Quantized networks and DL Boost support**
  If you plan to use quantized versions of neural networks, ensure that your CPU supports DL Boost. Without this feature, you may experience a significant drop in inference accuracy. In such cases, we recommend that you use the FP32 versions of the networks instead.
- **Processor requirements**
  Regardless of whether you are using quantized or non-quantized networks, we recommend that you use processors from the Intel Pentium Gold series or higher. These CPUs include advanced instruction sets like AVX512 FMA, which significantly enhance performance during network inference. When selecting a processor, prioritize models with a higher number of accelerators, as they directly impact computational efficiency.

11.3.2 GPU recommendations

For GPU-based inference, only server-grade (compute-class) GPUs are supported. Gaming GPUs are not recommended or supported for running LUNA SDK due to potential compatibility issues and performance limitations. Below is the list of supported GPUs:

Microarchitecture	Compute capability	GPU
Turing	7.5	Nvidia Tesla T4
Ampere	8.0	Nvidia A30
Ampere	8.0	Nvidia A100
Ampere	8.6	Nvidia A40
Ampere	8.6	Nvidia A10
Ampere	8.6	Nvidia A16
Ampere	8.6	Nvidia A2
Ada Lovelace	8.9	Nvidia L4
Ada Lovelace	8.9	Nvidia L40

11.4 Forking process

UNIX-like operating systems implement a mechanism to duplicate a process. It creates a new child process and copies its parents' memory space into the child's one. This is typically done programmatically by calling the `fork()` system function in the parent process.

Care should be taken when forking a process running the SDK.

> **Important:** Always fork before the first instance of `IFaceEngine` is created!
> This is because the SDK internally maintains a pool of worker threads, which is created lazily at the time the very first `IFaceEngine` object is born and destroyed right after the last `IFaceEngine` object is released. When using GPU or NPU devices, their runtime is initialized and shut down in the same manner.

The hazard comes from the fact that while `fork()` copies process memory, it only creates just one thread - the main thread. For details, see https://man7.org/linux/man-pages/man2/fork.2.html.

As a result, if at least one `IFaceEngine` object is alive at the time the process is being forked, the child processes will inherit the knowledge of the object, and therefore, the implicit thread pool (and device runtime, when appropriate). But there will be no worker threads actually running (in both, the inherited pool and the runtime, when appropriate) and attempting to call certain SDK functions will cause a deadlock.

11.5 Liveness estimator combination

Depending on your device and its camera, you can enhance the accuracy of the model by simultaneously using a combination of two universal liveness estimators. For example, you might use:

- LivenessDepthRGBEstimator and NIRLivenessEstimator
- LivenessDepthEstimator and LivenessOneShotRGBEstimator

To implement this, you need to aggregate the rates from each liveness estimator and adjust the thresholds in the faceengine.conf configuration file.

11.5.1 Changing the threshold

All models are calibrated so that the base threshold is 0.5 for any model of any modality.

If you need greater protection against hacking, then the threshold can be raised, and if the convenience of real users is more important, then lowered. We recommend that you configure specific values for changing the threshold in deviation from the basic one on a client basis.

11.5.2 Aggregating the scores

Any of two liveness modalities can be aggregated with each other. To do this, you need to multiply the speeds of the corresponding networks. The threshold in this case is also multiplied and becomes equal to 0.25.

11.5.3 Recommended thresholds

The recommended threshold is an optimal balance between TPR and FPR.

11.5.4 Possible LivenessOneShotRGBEstimator model combinations

You can use the LivenessOneShotRGBEstimator models in the following combinations:

For version `v12`:

- Use these models in the backend as an analogue of server LivenessOneShotRGBEstimator:
  - `oneshot_rgb_liveness_v12_model_1_cpu-avx2.plan`
  - `oneshot_rgb_liveness_v12_model_2_cpu-avx2.plan`
  - `oneshot_rgb_liveness_v12_model_3_cpu-avx2.plan`
  - `oneshot_rgb_liveness_v12_model_7_cpu-avx2.plan`
- Use these models on smartphones as an analogue of LivenessOneShotRGBEstimator:
  - `oneshot_rgb_liveness_v12_model_6_arm.plan`
- Use the below models on devices with Orbbec cameras, such as payment terminals (POS) and self-service cash registers (KCO):

- oneshot_rgb_liveness_v12_model_4_arm.plan
- oneshot_rgb_liveness_v12_model_5_arm.plan

- Use the following models for extended backend processing with higher accuracy requirements:
  - oneshot_rgb_liveness_v12_model_1_cpu-avx2.plan
  - oneshot_rgb_liveness_v12_model_2_cpu-avx2.plan
  - oneshot_rgb_liveness_v12_model_3_cpu-avx2.plan
  - oneshot_rgb_liveness_v12_model_7_cpu-avx2.plan
  - oneshot_rgb_liveness_v12_model_8_cpu-avx2.plan
  - oneshot_rgb_liveness_v12_model_9_cpu-avx2.plan
  - oneshot_rgb_liveness_v12_model_10_cpu-avx2.plan

For version v13:

- Use these models in the backend as an analogue of server LivenessOneShotRGBEstimator:
  - oneshot_rgb_liveness_v13_model_1_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_2_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_3_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_7_cpu-avx2.plan
- Use these models on smartphones as an analogue of LivenessOneShotRGBEstimator:
  - oneshot_rgb_liveness_v13_model_6_arm.plan
- Use the below models on devices with Orbbec cameras, such as payment terminals (POS) and self-service cash registers (KCO):
  - oneshot_rgb_liveness_v13_model_4_arm.plan
  - oneshot_rgb_liveness_v13_model_5_arm.plan
- Use the following models for extended backend processing with higher accuracy requirements:
  - oneshot_rgb_liveness_v13_model_1_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_2_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_3_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_7_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_8_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_9_cpu-avx2.plan
  - oneshot_rgb_liveness_v13_model_10_cpu-avx2.plan

12 Device-specific constraints

12.1 Image constraints

When memory is allocated for Image pixel data storage, the following constraints are enforced depending on the requested memory residence:

- Image::MemoryResidence::CPU: base address alignment is 32 bytes;
- Image::MemoryResidence::GPU: base address alignment is 128 bytes;
- Image::MemoryResidence::NPU: base address alignment is 128 bytes;
- Image::MemoryResidence::NPU_DPP: base address alignment is 128 bytes.

Also, in case of Image::MemoryResidence::NPU_DPP image width must be multiple of 16 and image height must be multiple of 2.

When Image is initialized as a wrapper for a user-provided memory block, whose residence is said to be Image::MemoryResidence::NPU or Image::MemoryResidence::NPU_DPP, the above requirements are checked upon the initialization.

Image class implements limited functionality for device-side data. Only the following operations are supported:

- construction (both with Image-owned memory and as a wrapper for a user-defined memory) and assignment (including deep copy);
- destruction;
- set() family of functions (functionally the same as construction/assignment);
- convert() function, but only in transfer mode; This means that both source and destination formats must match, only memory residency may differ. This function supports only synchronous memory transfers in the following directions:
  - host <-> GPU
  - GPU <-> GPU
  - host <-> NPU
  - NPU <-> NPU.

Full range of functionality (including format conversions) is currently only available for Images with host memory data residence.

The following operations are **NOT** supported:

- compressed format encoding/decoding;
- format/color space conversion;
- subimage views (i.e. map() function);
- padding and cropping (i.e. extract() function);
- manipulation (e.g. getPixel(), setPixel(), etc.).

13 Collecting information for Technical Support

To efficiently resolve a problem with LUNA SDK, collect all necessary information based on the error type and provide it to VisionLabs Technical Support. Possible error types include:

- Specific error
- Non-specific error
- Unexpected result

13.1 Contact Technical Support

You can contact our Technical Support in either of the following ways:

- Via email: support@visionlabs.ai
- Via Service Portal: https://jira.visionlabs.ru/servicedesk/customer/portal/2

13.2 Specific error

These errors usually occur when LUNA SDK is used incorrectly. Examples include:

- An estimator or detector does not work, resulting in an error when creating or using it.
- An error occurs when launching on a GPU device.
- A license error is received.

In such cases, study the full launch logs and understand what was launched and where.

To get detailed logging in LUNA SDK, follow these steps:

1. In the luna-sdk/data/runtime.conf configuration file, set the `verboseLogging` parameter to 4.

```
<param name="verboseLogging" type="Value::Int1" x="4" />
```

2. In the luna-sdk/data/faceengine.conf configuration file, set the `verboseLogging` parameter to 4.

```
<param name="verboseLogging" type="Value::Int1" x="4" />
```

3. In the luna-sdk/data/trackengine.conf configuration file, set the `severity` parameter to 0.

```
<param name="severity" type="Value::Int1" x="0" />
```

If you know which module the error occurs in, provide only that module's log by changing the value only in the relevant configuration file. If unsure, collect all logs.

13.3 Non-specific error

Examples of non-specific errors include:

- An application crashes at an uncertain time.
- An application freezes unexpectedly.
- There is a memory leak.

In such cases, you need to understand in detail the application operation scenario, including what is called and in what sequence.

Provide the following information:

- The exact version of LUNA SDK (e.g., v.5.22.2, build for CentOS 8).
- Information about the environment where the application runs (e.g., Docker container, launch via Python bindings).
- Full launch logs.
- Additional information like crash dumps, reports from third-party utilities, and system logs.
- Code reproducing the problem, if any.

13.4 Unexpected Result

Unexpected results may occur due to:

- Incorrect use of LUNA SDK
- Algorithm errors
- Launching in unexpected conditions

Examples include:

- A face is present in a photo or video, but the detector doesn't see it.
- A person is smiling, but the emotion estimator indicates sadness.

Reasons for unexpected results vary, such as:

- Incorrect use of LUNA SDK, for example, a wrong threshold in a configuration file.
- Incorrect input data, such as a poor-quality video or heavily compressed frames.
- Occasional algorithm errors.
- New data for the algorithm.

To understand and address the issue, provide:

- Full launch logs.
- All configuration files used during the launch:
  - luna-sdk/data/runtime.conf
  - luna-sdk/data/faceengine.conf
  - luna-sdk/data/trackengine.conf

- An estimate of how often the unexpected result occurs, for example, every frame or once in a thousand frames.
- Examples of data that produce unexpected results.

14 Useful tools

14.1 Performance testing

Performance testing is crucial for ensuring the reliability, accuracy, and efficiency of software systems. It helps in optimizing resource usage, reducing latency, and providing consistent results across different environments. Below are key concepts, metrics, parameters, and practical recommendations for conducting effective performance tests.

14.1.1 Key concepts in performance testing

- **Warm-up Phase**
  Initial iterations often include delays due to memory allocation, lazy data initialization, thread creation, and caching. These effects diminish after a few iterations. Warm-up iterations are excluded from final results to ensure accuracy.
- **Noise Compensation**
  Noise in performance tests arises from factors like OS multitasking, resource contention, and memory management. To mitigate noise:
  - Increase the number of iterations to average out high-frequency noise.
  - Use statistical methods such as averaging or filtering to stabilize results.

14.2 Metrics for performance analysis

14.2.1 Common metrics

Metric	Description
`min`	The smallest measured time across all iterations. It is protected by hardware limitations, less sensitive to anomalies compared to `max`, `avg`, or `median` and does not reflect worst-case scenarios.
`max`	The largest measured time across all iterations. It reflects extreme cases (for example, system delays) and is highly variable between runs, no upper boundary, sensitive to OS delays.
`avg`	The arithmetic mean of all measured times. It is simple to calculate and sensitive to outliers; a single large value can significantly increase the average.

Metric	Description
`median`	The middle value in the sorted list of measured times. It is more robust than `avg` but less reliable than `min` and resistant to moderate anomalies. It can shift upward if multiple anomalies fall into the upper half of the sorted list.
`mode`	The most frequently occurring value in the measured times and the most reliable metric for analysis, unaffected by rare anomalies, works well with asymmetric distributions. It requires careful histogram construction to avoid instability.

14.2.1.1 Practical use

- Use `min` for determining convergence because it approaches a hardware-determined lower bound as iterations increase.
- Combine metrics for comprehensive analysis (for example, `min` for stability, `max` for outliers).

14.3 Performance test parameters

Below are additional command line parameters that allow you to customize performance test operation.

14.3.1 Test-specific parameters

Parameter	Description
`-t, --test`	Specifies the type of test being performed (mandatory named parameter).
`-i, --image`	Specifies the input image for tests (named parameter).
`-o, --out`	Specifies the output CSV file for final statistics (mandatory named parameter).
`--raw-out`	Specifies the CSV file for recording operational statistics after each iteration. Includes all iterations, even those during warm-up.

14.3.2 Batch and sensor parameters

Parameter	Description
`-b, --batch`	Sets the batch size (named parameter).
`-s, --sensor`	Sets the sensor type, for example, for the EyesBatch test.
`-y, --yuv`	Sets the YUV image for *YUV12toRGB* and *YUV21toRGB* tests.

14.3.3 iOS-specific parameters

Parameter	Description
`--data`	Path to the data directory (used only in non-standard iOS mode).
`--threads`	Number of threads used for testing (used only in non-standard iOS mode).
`--descriptor-model`	Specifies the descriptor model used in tests.
`--detector-type`	Specifies the detector type used in tests.

14.3.4 Stopping condition parameters

Parameter	Description
`--max-rel-height`	Threshold for relative height of the last step. If exceeded, stopping conditions are not met.
`--min-step-width`	Minimum width of the last step. If narrower, stopping conditions are not met.
`--max-rel-slope`	Threshold for relative slope of the last step. Combines the effects of `--max-rel-height` and `--min-step-width`.
`--min-steps`	Minimum number of steps required before stopping conditions can be evaluated.
`--min-iters`	Minimum number of iterations required before stopping.

Parameter	Description
`--max-iters`	Maximum number of iterations allowed (emergency stop condition).
`--max-time`	Maximum total execution time allowed (emergency stop condition).

14.3.5 Recommendations for parameter selection

- Start with default parameters.
  Avoid overriding these settings unless necessary, as doing so may unnecessarily extend the execution time of most tests.
- Optimize runtime.
  If the test runtime is excessively long, consider relaxing the thresholds for the following parameters:
  - `--min-step-width`
  - `--max-rel-height`
  - `--max-rel-slope`
  - `--min-steps`
  - `--min-iters`

  Adjusting these parameters will cause the convergence-based stopping conditions to trigger more quickly, thereby reducing the overall test duration. However, this approach may compromise the reliability and stability of the results.
- Balance runtime and result quality.
  Striking a balance between runtime efficiency and result quality is one of the key trade-offs when configuring a performance test. While loosening thresholds can expedite the test, it is essential to ensure that the resulting data remains sufficiently accurate and stable for meaningful analysis.

14.4 Stopping conditions

14.4.1 Normal stopping conditions

- **Convergence analysis:**

  The test analyzes the convergence of `min` values over iterations.

  Key metrics:

  - `step_height` - Absolute change in `min` between two consecutive iterations.
  - `rel_step_height` - Relative change in `min` as a percentage of the previous value.
  - `step_width` - Number of consecutive iterations where `min` does not improve.

- `rel_slope` - Rate of change in `min` per iteration.

Conditions:

- If `rel_step_height` is below a threshold (`--max-rel-height`), convergence is assumed.
- If `step_width` exceeds a threshold (`--min-step-width`), it indicates that further improvements require too few iterations.
- If `rel_slope` is below a threshold (`--max-rel-slope`), it confirms slow changes in `min`.

- **Minimum steps/iterations :**
  - At least `--min-steps` must be generated to ensure stability.
  - At least `--min-iters` iterations must be completed to ensure sufficient data collection.

14.4.2 Emergency stopping conditions

- **Exceeding iteration limit:**
  If the number of iterations reaches `--max-iters`, the test stops regardless of convergence.
- **Exceeding time limit:**
  If the total execution time exceeds `--max-time`, the test stops regardless of convergence.
- **Insufficient warm-up:**
  If an emergency stop occurs before completing the warm-up phase, results may be unreliable due to incomplete stabilization of initial delay.

14.4.2.1 Configuration of emergency stop conditions

The `--max-iters` and `--max-time` parameters are designed to trigger an emergency stop of the test. These safeguards prevent the performance test from running indefinitely in cases where convergence issues arise.

A normal stop should occur before these emergency thresholds are reached. Ideally, the test will meet its convergence criteria and terminate well before approaching the emergency limits. To ensure this, we recommend that you set `--max-iters` and `--max-time` with a generous margin, so they significantly exceed the expected duration for a successful, routine stop.

By doing so, you can avoid premature terminations due to overly restrictive settings and allow the test sufficient time to achieve stable results under normal conditions.

14.4.3 Special cases

- **Local minima:**
  If small steps are formed early in the test, the `--min-steps` parameter ensures enough steps are generated to confirm global convergence.
- **Last iteration uncertainty:**
  For the final iteration, future behavior is unknown, so no step parameters are defined.

14.5 Example console report

During a performance test execution, the console displays operational statistics that help you track the current test results. Operational statistics show all iterations, including warm-up iterations. Here is how it is organized:

```
Performing test.
width[iters] rel.height[%] rel.slope[%/iter] min[ms]   max[ms]   avg[ms]
------------ ------------- ----------------- --------- --------- ---------
0            N/A           N/A               3.247     3.247     3.247
1            13.735752     13.735752         2.801     3.247     3.0240002
.
2            3.355948      1.677974          2.707     3.247     2.9437501
1            3.5094209     3.5094209         2.612     3.247     2.8774002
1            7.924962      7.924962          2.405     3.247     2.798667
1            0.7900231     0.7900231         2.386     3.247     2.7397144
...
4            1.6764443     0.41911107        2.346     3.247     2.671273
1            1.4066494     1.4066494         2.313     3.247     2.6414168
........................
25           2.8534365     0.114137456       2.247     3.247     2.5652163
.............
14           4.6283975     0.3305998         2.143     3.247     2.5182943
..............................................................
63           2.0065322     0.03184972        2.1       3.247     2.4318075
.........................................................................
74           0.95238006    0.012870001       2.08      3.247     2.399947
1            N/A           N/A               2.08      3.247     2.399947

Break condition = converged
188 total runs, 62 warmup runs, batch size = 32
AGSBatch execution time:
                 total          296.978 ms
                 avg             per batch: 2.37 ms               per image: 0.07405 ms
                 max             per batch: 2.88 ms               per image: 0.09 ms
                 min             per batch: 2.08 ms               per image: 0.065 ms
                 std             +/- 14 %
                 50%             per batch: 2.338 ms              per image: 0.07306 ms
                 90%             per batch: 2.543 ms              per image: 0.07947 ms
                 95%             per batch: 2.613 ms              per image: 0.08166 ms
                 mode            per batch: [2.270,2.292) ms      per image: [0.071,0.072) ms
```

Figure 24: Performance test console report

By analyzing the console report, you can assess the stability of results, identify potential issues, and ensure the test converges correctly before relying on the final output.

14.5.1 Structure of the first table

Each row in the table represents one "step" or "staircase" of the min value over time.

Between steps, there may be idle iterations that do not improve the `min` value; these are marked with dots (.), one for each iteration.

14.5.2 Column contents

- **First column:** Step width (number of consecutive iterations). The sum of all step widths equals the total number of iterations, including the warm-up phase.
- **Second column:** Relative height of the step.
- **Third column:** Relative slope of the step (percentage change per iteration).

14.5.3 Additional metrics

For every generated step, three metrics are displayed:

Metric	Description
`min`	Current minimum time after the current iteration.
`max`	Maximum time across all completed iterations.
`avg`	Average time across all completed iterations.

14.5.4 Zero and last iterations

- **Zero iteration:** Parameters for the first step are undefined because no prior `min` values exist.
- **Last iteration:** Parameters for the final step are also undefined since it is unknown whether further iterations would have reduced `min`.

14.5.5 Color coding

Changes in relative height or slope compared to the previous step are color-coded:

- Green indicates an increase in relative height or slope.
- Red indicates a decrease in relative height or slope.

14.5.6 Reasons for stopping

After the table, the console displays the reason for stopping the test:

- **Normal stop:** Conditions for convergence were met.
- **Emergency stop:** Exceeded `--max-time` or `--max-iters`.

14.5.7 Operational vs. final statistics

Operational statistics	Final statistics
Includes all iterations, even those during the warm-up phase.	Excludes warm-up iterations and focuses on post-warm-up data and adds the calculation of mode (most frequent value) for better accuracy.
Values like `max` and `avg` in operational statistics tend to be higher than in final statistics due to the inclusion of warm-up data.	Warm-up iterations account for initial delays ($d(t)$), which skew early results but are excluded from final reports.
Operational `max` includes warm-up delays, so it appears higher.	Final `max` excludes warm-up, providing a more accurate representation of steady-state performance.

14.6 Performance test challenges

14.6.1 Measurement range limitations

Performance tests are unsuitable for measuring time intervals in the range of a few nanoseconds to several hundred microseconds. For such cases, use microbenchmark frameworks. However, performance tests excel at measuring time in the required range — from milliseconds and above.

14.6.2 High-frequency noise

Performance tests effectively filter out high-frequency noise, such as random delays with periods much shorter than the total execution time of the test across all iterations of one type.

14.6.3 Low-frequency noise

Performance tests cannot efficiently handle low-frequency noise if its characteristic duration is comparable to or exceeds the test execution time. For example:

- Delays during the warm-up phase (d(t)), which are predictable and easily compensated.
- Service processes (updates, defragmentation, backups) running for several hours. If the test runtime overlaps with these processes, results will be distorted.
- Low-frequency noise affects all types of measurements. Collecting long-term statistics to detect and filter such noise is often impractical or too resource-intensive. Therefore, minimizing its impact relies on user intervention, such as proper server configuration.

14.6.4 Test execution duration

The primary new challenge for performance tests is the significant amount of time required to gather a sufficient sample of data.

14.6.5 Artificial constraints efficiency

Artificial constraints via `--max-iters` or `--max-time` reduce the test's effectiveness by limiting the dataset size, potentially compromising reliability.

14.6.6 Launch recommendations

- Run tests overnight when system load is minimal for optimal results.
- Daytime runs can be conducted with reduced execution time for quick analysis but should be treated as preliminary, as they may lack accuracy.

14.7 Potential improvements

These improvements aim to streamline the testing process, provide deeper insights, and reduce manual intervention, ultimately resulting in more efficient and accurate performance evaluations:

- **Automatic chart generation**
  Utilize libraries like Plotly to create visually appealing and interactive web-based charts. This enhances clarity, simplifies analysis, and improves usability.
- **Continuous function approximation**
  Instead of discrete histograms, approximate measurement distribution using continuous functions. This eliminates issues related to bin size and count, improving accuracy.
- **Enhanced warm-Up logic**
  Dynamically calculate the required number of warm-up iterations based on specific test needs, improving both accuracy and efficiency.
- **Automated result comparison**
  Implement an automated system to compare current results with previous runs, generating reports on performance improvements or regressions. Visualizing changes through graphs and detecting abnormal performance drops would enhance responsiveness to issues.
- **Advancements in convergence analysis**
  Refine algorithms for detecting stabilized metrics, incorporate advanced statistical methods to handle noise and outliers, and improve heuristics for identifying global versus local minima during the test.

14.8 Practical recommendations

- Always include a warm-up phase to eliminate initialization delays from results.
- Use a sufficient number of iterations to reduce noise and achieve stable metrics.
- Focus on the `min` metric for determining convergence due to its stability and predictable behavior.
- Visualize results using tools like CSV exports and graphs for better interpretation of trends and anomalies.

15 Appendix A. Specifications

15.1 Classification performance

Classification performance was measured on a two datasets:

- Cooperative dataset (containing 20K images from various sources obtained at several banks);
- Non cooperative dataset (containing 20K).

The two tables below contain true positive rates corresponding to select false positive rates.

Table 77: "Classification performance @ low FPR on cooperative dataset"

FPR	TPR CNN 58	TPR CNN 59	TPR CNN 59m	TPR CNN 60	TPR CNN 60m	TPR CNN 62	TPR CNN 65
10^{-7}	0.9910	0.9911	0.9809	0.9917	0.979	0.9916	0.9909
10^{-6}	0.9916	0.9915	0.9876	0.9917	0.989	0.9917	0.9950
10^{-5}	0.9918	0.9919	0.9904	0.9919	0.990	0.9918	0.9976
10^{-4}	0.9919	0.9921	0.9915	0.9921	0.991	0.9920	0.9988

Table 78: "Classification performance @ low FPR on non cooperative dataset"

FPR	TPR CNN 58	TPR CNN 59	TPR CNN 59m	TPR CNN 60	TPR CNN 60m	TPR CNN 62	TPR CNN 65
10^{-7}	0.9834	0.9850	0.9059	0.9862	0.9279	0.9909	0.9909
10^{-6}	0.9914	0.9907	0.9454	0.9931	0.9523	0.9950	0.9950
10^{-5}	0.9954	0.9956	0.9705	0.9967	0.9752	0.9976	0.9976
10^{-4}	0.9983	0.9983	0.9868	0.9987	0.9888	0.9988	0.9988

15.2 Runtime performance for CentOS Linux environment

Face detection performance depends on input image parameters, including resolution, bit depth, and the size of the detected face.

Input data characteristics:

- Image resolution: 1920x1080px
- Image format: 24 BPP RGB

Performance measurements for CPU, GPU, and NPU execution modes are presented in the tables below. The measured values represent averages from at least 100 experiments.

Estimated memory consumption values are also provided for CPU and GPU. These values are highly dependent on the input data and the experimental conditions.

The results for both minimum and optimal batch sizes are shown in the tables below, while all intermediate and non-optimal values have been omitted.

Face detection is performed using the FaceDetV3 neural network. All types of face detection and re-detection include capturing bounding boxes and five facial landmarks.

15.2.1 CPU performance

Benchmarking for CPU was performed on a server with the following hardware configuration:

CPU:

- Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz;
- CPU(s): 40
- Thread(s) per core: 2
- Core(s) per socket: 10
- Socket(s): 2
- NUMA node(s): 2
- CPU with AVX2 instruction set was used

OS: CentOS Linux release 8.3.2011

RAM: 128 GB DDR4 (Clock Speed: 2133 MHz)

In experiments listed in tables below, face detection and descriptor extraction algorithms used all available CPU cores, while the matching performance is specified on a per-core basis.

Descriptor matching is only implemented on the CPU.

15.2.1.1 CPU. Detector performance

The table below shows the performance of FaceDetV3 Detector on the CPU.

Measurement	CPU threads	BatchSize	Percentile 95 (ms)	RAM Memory (Mb)
Detector (minFaceSize=20)	1	1	339.83	2005.0
Detector (minFaceSize=20)	8	1	116.0	2154.0
Detector (minFaceSize=20)	8	8	111.1	5385.0
Detector (minFaceSize=50)	1	1	56.25	1616.0

Measurement	CPU threads	BatchSize	Percentile 95 (ms)	RAM Memory (Mb)
Detector (minFaceSize=50)	8	1	22.74	1707.0
Detector (minFaceSize=50)	8	8	19.04	2356.0
Detector (minFaceSize=90)	1	1	20.58	1566.0
Detector (minFaceSize=90)	8	1	10.07	1628.0
Detector (minFaceSize=90)	8	8	6.79	1897.0
Redetect	1	1	0.65	1609.0
Redetect	8	1	0.79	1651.0
Redetect	8	8	0.23	2223.0
Landmarks5Detector	1	1	0.22	1614.0
Landmarks5Detector	8	1	0.29	1639.0
Landmarks5Detector	8	8	0.08	1642.0
Landmarks68Detector	1	1	4.08	1619.0
Landmarks68Detector	8	1	2.08	1639.0
Landmarks68Detector	8	8	1.11	1650.0

15.2.1.2 CPU. HumanDetector performance

The table below shows the performance of HumanDetector on the CPU.

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanDetector (resize to 320)	1	1	10.38	1542.0
HumanDetector (resize to 320)	8	1	5.98	1590.0
HumanDetector (resize to 320)	8	8	3.51	1817.0
HumanDetector (resize to 640)	1	1	36.18	1573.0
HumanDetector (resize to 640)	8	1	14.48	1631.0
HumanDetector (resize to 640)	8	8	11.66	2019.0
HumanRedetect	1	1	2.61	1572.0
HumanRedetect	8	1	2.4	1632.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanRedetect	8	8	1.11	1941.0
HumanWarper	1	1	0.35	1536.0
HumanWarper	8	1	0.4	1544.0
HumanWarper	8	8	0.12	1582.0
HumanWarper	1	1	0.39	1558.0
HumanWarper	8	1	0.4	1584.0
HumanWarper	8	8	0.12	1622.0

15.2.1.3 CPU. HumanFaceDetector performance

The table below shows the performance of HumanFaceDetector on CPU with the AVX2 plan-file (useInt8=0)

Measurement	CPU threads	BatchSize	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	1	1	377.51	2356.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	8	1	103.4	2635.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	8	8	99.2	6324.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	1	1	61.68	1925.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	8	1	21.81	2085.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	8	8	15.98	2783.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	1	1	22.87	1872.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	8	1	10.04	2003.0

Measurement	CPU threads	BatchSize	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	8	8	5.78	2364.0
HumanFaceDetectorBoxes (minFaceSize=20)	1	1	369.35	2336.0
HumanFaceDetectorBoxes (minFaceSize=20)	8	1	99.18	2616.0
HumanFaceDetectorBoxes (minFaceSize=20)	8	8	93.45	5976.0
HumanFaceDetectorBoxes (minFaceSize=50)	1	1	61.28	1930.0
HumanFaceDetectorBoxes (minFaceSize=50)	8	1	20.88	2119.0
HumanFaceDetectorBoxes (minFaceSize=50)	8	8	15.43	2745.0
HumanFaceDetectorBoxes (minFaceSize=90)	1	1	22.34	1876.0
HumanFaceDetectorBoxes (minFaceSize=90)	8	1	9.75	1995.0
HumanFaceDetectorBoxes (minFaceSize=90)	8	8	5.45	2284.0

The table below shows the performance of HumanFaceDetector on CPU with the AVX2-INT8 plan-file (useInt8=1)

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	1	1	350.14	1926.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	8	1	132.29	2028.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	8	8	127.61	4650.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	1	1	64.5	1627.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	8	1	26.7	1694.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	8	8	20.11	2210.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	1	1	25.85	1583.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	8	1	11.85	1654.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	8	8	7.38	1878.0
HumanFaceDetectorBoxes (minFaceSize=20)	1	1	330.24	1896.0
HumanFaceDetectorBoxes (minFaceSize=20)	8	1	117.24	2001.0
HumanFaceDetectorBoxes (minFaceSize=20)	8	8	106.87	4388.0
HumanFaceDetectorBoxes (minFaceSize=50)	1	1	65.7	1619.0
HumanFaceDetectorBoxes (minFaceSize=50)	8	1	26.32	1691.0
HumanFaceDetectorBoxes (minFaceSize=50)	8	8	19.33	2149.0
HumanFaceDetectorBoxes (minFaceSize=90)	1	1	26.21	1581.0
HumanFaceDetectorBoxes (minFaceSize=90)	8	1	12.2	1645.0
HumanFaceDetectorBoxes (minFaceSize=90)	8	8	7.22	1871.0

15.2.1.4 CPU. HeadDetector performance

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadDetector (minHeadSize=20)	1	1	340.62	1992.0
HeadDetector (minHeadSize=20)	8	1	115.54	2151.0
HeadDetector (minHeadSize=20)	8	8	111.67	5380.0
HeadDetector (minHeadSize=50)	1	1	55.13	1610.0
HeadDetector (minHeadSize=50)	8	1	22.28	1690.0
HeadDetector (minHeadSize=50)	8	8	18.86	2348.0
HeadDetector (minHeadSize=90)	1	1	20.85	1558.0
HeadDetector (minHeadSize=90)	8	1	9.7	1648.0
HeadDetector (minHeadSize=90)	8	8	6.79	1909.0

15.2.1.5 CPU. Estimations performance with batch interface

The table below shows the performance of Estimations on the CPU for estimators that have a batch interface. All these measurements are performed with `minFaceSize=50`.

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadPose	1	1	0.26	1646.0
HeadPose	8	1	0.17	1801.0
HeadPose	8	8	0.07	1840.0
Warper	1	1	2.08	1653.0
Warper	8	1	2.35	1804.0
Warper	8	8	0.57	1803.0
Eyes (RGB, useStatusPlan=0)	1	1	1.4	1648.0
Eyes (RGB, useStatusPlan=0)	8	1	0.43	1795.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Eyes (RGB, useStatusPlan=0)	8	8	0.23	1800.0
Eyes (RGB, useStatusPlan=1)	1	1	1.27	1643.0
Eyes (RGB, useStatusPlan=1)	8	1	0.43	1788.0
Eyes (RGB, useStatusPlan=1)	8	8	0.23	1800.0
Eyes (INFRA RED, useStatusPlan=0)	1	1	0.8	1557.0
Eyes (INFRA RED, useStatusPlan=0)	8	1	0.37	1632.0
Eyes (INFRA RED, useStatusPlan=0)	8	8	0.18	1634.0
Eyes (INFRA RED, useStatusPlan=1)	1	1	0.8	1565.0
Eyes (INFRA RED, useStatusPlan=1)	8	1	0.38	1630.0
Eyes (INFRA RED, useStatusPlan=1)	8	8	0.19	1634.0
InfraRed	1	1	2.06	1562.0
InfraRed	8	1	0.94	1641.0
InfraRed	8	8	0.72	1640.0
AGS	1	1	0.25	1641.0
AGS	8	1	0.17	1788.0
AGS	8	8	0.08	1844.0
BestShotQuality	1	1	0.46	1652.0
BestShotQuality	8	1	0.22	1806.0
BestShotQuality	8	8	0.1	1837.0
MedicalMask	1	1	5.11	1962.0
MedicalMask	8	1	3.1	2034.0
MedicalMask	8	8	0.96	2075.0
LivenessOneShotRGBEstimator 2XL	1	1	1297.64	2842.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
LivenessOneShotRGBEstimator 2XL	8	1	475.71	3042.0
LivenessOneShotRGBEstimator 2XL	8	8	615.84	4263.0
LivenessOneShotRGBEstimator XL	1	1	304.3	1881.0
LivenessOneShotRGBEstimator XL	8	1	82.44	2109.0
LivenessOneShotRGBEstimator XL	8	8	61.41	2612.0
Orientation	1	1	6.89	1543.0
Orientation	8	1	3.82	1567.0
Orientation	8	8	2.13	1615.0
FacialHair	1	1	13.72	1663.0
FacialHair	8	1	4.5	1820.0
FacialHair	8	8	3.96	1806.0
CredibilityCheck	1	1	124.03	1765.0
CredibilityCheck	8	1	34.23	1896.0
CredibilityCheck	8	8	34.47	2181.0
BlackWhite	1	1	1.28	1650.0
BlackWhite	8	1	0.5	1816.0
BlackWhite	8	8	0.48	1803.0
NaturalLight	1	1	2.21	1650.0
NaturalLight	8	1	1.4	1799.0
NaturalLight	8	8	0.73	1806.0
PortraitStyle	1	1	1.01	1639.0
PortraitStyle	8	1	1.05	1792.0
PortraitStyle	8	8	0.44	1841.0
FishEye	1	1	2.29	1653.0
FishEye	8	1	1.38	1807.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
FishEye	8	8	0.95	1823.0
EyeBrow	1	1	13.5	1662.0
EyeBrow	8	1	4.42	1821.0
EyeBrow	8	8	3.88	1807.0
HumanAttribute	1	1	12.79	1565.0
HumanAttribute	8	1	6.02	1589.0
HumanAttribute	8	8	3.77	1648.0
RedEye	1	1	2.64	1648.0
RedEye	8	1	0.87	1791.0
RedEye	8	8	0.79	1800.0
HeadWear	1	1	4.49	1657.0
HeadWear	8	1	2.63	1816.0
HeadWear	8	8	1.19	1822.0
Background	1	1	1.04	1644.0
Background	8	1	1.03	1794.0
Background	8	8	0.44	1859.0
Mouth	1	1	6.88	1657.0
Mouth	8	1	2.55	1810.0
Mouth	8	8	2.1	1801.0
Attributes (netType=0, precise)	1	1	62.5	1704.0
Attributes (netType=0, precise)	8	1	19.5	1843.0
Attributes (netType=0, precise)	8	8	18.0	2106.0
Attributes (netType=1, fast)	1	1	7.4	1665.0
Attributes (netType=1, fast)	8	1	3.5	1814.0
Attributes (netType=1, fast)	8	8	2.2	1826.0
Subjective Quality	1	1	1.41	1946.0
Subjective Quality	8	1	0.63	2009.0
Subjective Quality	8	8	0.29	2016.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Emotions	1	1	13.53	1662.0
Emotions	8	1	4.59	1819.0
Emotions	8	8	3.88	1814.0
EyesGaze	1	1	2.32	1656.0
EyesGaze	8	1	1.3	1801.0
EyesGaze	8	8	0.65	1794.0
Glasses	1	1	0.93	1647.0
Glasses	8	1	0.95	1786.0
Glasses	8	8	0.4	1800.0
LivenessFlyingFaces	1	1	14.89	1680.0
LivenessFlyingFaces	8	1	6.55	1878.0
LivenessFlyingFaces	8	8	4.78	1989.0
DynamicRange	1	1	1.37	1643.0
DynamicRange	8	1	1.69	1799.0
DynamicRange	8	8	0.39	1831.0
Ethnicity	1	1	13.24	1665.0
Ethnicity	8	1	4.45	1812.0
Ethnicity	8	8	3.87	1808.0
DeepFake	1	1	404.49	2210.0
DeepFake	8	1	79.09	2574.0
DeepFake	8	8	75.51	4137.0
Fights	1	1	230.46	1798.0
Fights	8	1	58.68	1825.0
NIRLivenessEstimator	1	1	17.1	1540.0
NIRLivenessEstimator	8	1	11.32	1560.0
NIRLivenessEstimator	8	8	10.64	1663.0
LivenessRGBMEstimator	1	1	27.83	1661.0
LivenessRGBMEstimator	8	1	10.52	1830.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
LivenessRGBMEstimator	8	8	8.53	2163.0
DepthLivenessEstimator	1	1	2.03	1524.0
DepthLivenessEstimator	8	1	1.23	1547.0
DepthLivenessEstimator	8	8	0.83	1569.0
YUV12toRGB	1	1	6.26	112.0
YUV12toRGB	8	1	6.33	112.0
YUV12toRGB	8	8	6.26	111.0
YUV21toRGB	1	1	6.71	110.0
YUV21toRGB	8	1	6.76	111.0
YUV21toRGB	8	8	6.72	112.0
Rotation	1	1	12.02	120.0
Rotation	8	1	11.97	118.0
FaceOcclusion	1	1	7.58	1664.0
FaceOcclusion	8	1	3.51	1807.0
FaceOcclusion	8	8	2.94	1819.0
ImageModification	1	1	12.31	1557.0
ImageModification	8	1	5.75	1572.0
ImageModification	8	8	3.87	1679.0

15.2.1.6 CPU. Estimations performance without batch interface

The table below shows the performance of Estimations on the CPU for estimators that do not have a batch interface. All these measurements are performed with `minFaceSize=50`.

Measurement	CPU threads	Percentile 95 (ms)	RAM Memory (Mb)
PPE	8	5.78	1602.0
PPE	1	12.17	1584.0
Overlap	8	1.22	1636.0
Overlap	1	4.83	1612.0

15.2.1.7 CPU. Extractor performance

The table below shows the performance of Extractor on the CPU.

Model	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
58	1	1	213.91	1814.0
58	8	1	57.36	1835.0
58	8	8	61.0	1950.0
59	1	1	214.29	1810.0
59	8	1	56.84	1823.0
59	8	8	60.93	1957.0
60	1	1	215.24	1803.0
60	8	1	57.74	1830.0
60	8	8	61.25	1944.0
62	1	1	256.36	1866.0
62	8	1	65.44	1885.0
62	8	8	72.15	1985.0
65	1	1	341.79	1960.0
65	8	1	99.06	1971.0
65	8	8	100.14	2816.0
105	1	1	1.66	1604
105	8	8	0.71	1657
106	1	1	140.76	1892
106	8	8	39.01	1954
107	1	1	12.0	1637
107	8	8	3.7	1723
108	1	1	2.41	1522.0
108	8	1	2.27	1541.0
108	8	8	0.81	1598.0
109	1	1	133.7	1822
109	8	8	37.33	1889
110	1	1	15.53	1640

Model	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
110	8	8	5.39	1733
112	1	1	118.69	1713.0
112	8	1	42.07	1727.0
112	8	8	34.05	1802.0
113	1	1	15.85	1553.0
113	8	1	6.8	1576.0
113	8	8	4.72	1633.0
115	1	1	119.36	1715.0
115	8	1	40.21	1736.0
115	8	8	34.06	1803.0
116	1	1	16.86	1550.0
116	8	1	7.2	1570.0
116	8	8	4.97	1638.0

15.2.1.8 CPU. Matcher performance

The table below shows the performance of Matcher on the CPU. The table includes average matcher per second for descriptors received using the following CNN model versions:

Model	CPU threads	Batch Size	PerSecond	RAM Memory (Mb)
58	1	1000	41163.9	97.0
59	1	1000	41580.1	101.0
60	1	1000	41386.2	97.0
62	1	1000	41309.8	98.0
65	1	1000	41340.0	98.0
108	1	1000	41220.8	98.0
112	1	1000	41652.3	97.0
113	1	1000	41796.1	97.0
115	1	1000	41408.9	97.0
116	1	1000	41487.4	99.0

> **Note:** The value above represents the maximum performance of the matcher on a specific piece of hardware. Overall performance does not depend on batch size; however, it may be limited by memory performance when using large batch sizes.

15.2.1.9 CPU. CrowdEstimator performance

The table below shows the performance of CrowdEstimator on the CPU.

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
CrowdEstimator (Single, minHeadSize=6)	1	1	3110.01	2488.0
CrowdEstimator (Single, minHeadSize=6)	8	1	922.77	2490.0
CrowdEstimator (Single, minHeadSize=6)	8	8	611.5	8649.0
CrowdEstimator (Single, minHeadSize=12)	1	1	774.76	1826.0
CrowdEstimator (Single, minHeadSize=12)	8	1	235.78	1844.0
CrowdEstimator (Single, minHeadSize=12)	8	8	139.94	3395.0
CrowdEstimator (TwoNets, minHeadSize=6)	1	1	3185.38	2501.0
CrowdEstimator (TwoNets, minHeadSize=6)	8	1	930.5	2679.0
CrowdEstimator (TwoNets, minHeadSize=6)	8	8	636.19	9149.0
CrowdEstimator (TwoNets, minHeadSize=12)	1	1	787.35	1854.0
CrowdEstimator (TwoNets, minHeadSize=12)	8	1	237.6	1977.0
CrowdEstimator (TwoNets, minHeadSize=12)	8	8	146.16	3790.0

15.2.2 GPU performance

Benchmarking for the GPU was performed on the following hardware configuration:

GPU: NVIDIA Tesla T4.

OS: CentOS Linux release 8.3.2011

15.2.2.1 GPU. Detector performance

The table below shows the performance of FaceDetV3 Detector on the GPU.

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
Detector (minFaceSize=20)	1	24.93	1436.0	1674.0
Detector (minFaceSize=20)	4	29.52	3946.0	1687.0
Detector (minFaceSize=20)	8	32.65	7338.0	1742.0
Detector (minFaceSize=50)	1	7.01	712.0	1664.0
Detector (minFaceSize=50)	4	5.78	1242.0	1696.0
Detector (minFaceSize=50)	8	5.42	1806.0	1717.0
Detector (minFaceSize=90)	1	4.66	624.0	1658.0
Detector (minFaceSize=90)	4	3.11	780.0	1681.0
Detector (minFaceSize=90)	8	2.85	978.0	1700.0
Redetect	1	2.3	712.0	1624.0
Redetect	8	0.29	1758.0	1641.0
Redetect	16	0.23	2834.0	1674.0
Landmarks5Detector	1	2.13	712.0	1670.0
Landmarks5Detector	8	0.31	712.0	1670.0
Landmarks5Detector	16	0.2	712.0	1678.0
Landmarks68Detector	1	2.38	712.0	1610.0
Landmarks68Detector	8	0.54	744.0	1671.0
Landmarks68Detector	16	0.25	744.0	1612.0

15.2.2.2 GPU. HumanDetector performance

The table below shows the performance of HumanDetector on the GPU.

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
HumanDetector (resize to 320)	1	3.99	598.0	1606.0
HumanDetector (resize to 320)	4	2.59	670.0	1610.0
HumanDetector (resize to 320)	8	2.16	892.0	1641.0
HumanDetector (resize to 640)	1	6.41	646.0	1594.0
HumanDetector (resize to 640)	4	4.04	864.0	1624.0
HumanDetector (resize to 640)	8	3.83	1222.0	1651.0
HumanRedetect	1	2.59	646.0	1661.0
HumanRedetect	8	0.41	1206.0	1664.0
HumanRedetect	16	0.21	1778.0	1665.0
HumanWarper	1	0.05	604.0	1592.0
HumanWarper	4	0.03	622.0	1609.0
HumanWarper	8	0.03	648.0	1621.0
HumanWarper	1	0.04	652.0	1591.0
HumanWarper	4	0.03	670.0	1608.0
HumanWarper	8	0.03	696.0	1621.0

15.2.2.3 GPU. HeadDetector performance

The table below shows the performance of HeadDetector on the GPU.

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
HeadDetector (minHeadSize=20)	1	24.61	1436.0	1614.0
HeadDetector (minHeadSize=20)	4	29.29	3978.0	1637.0
HeadDetector (minHeadSize=20)	8	34.19	7366.0	1681.0
HeadDetector (minHeadSize=50)	1	6.36	712.0	1602.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
HeadDetector (minHeadSize=50)	4	5.64	1242.0	1625.0
HeadDetector (minHeadSize=50)	8	5.34	1806.0	1656.0
HeadDetector (minHeadSize=90)	1	4.01	624.0	1602.0
HeadDetector (minHeadSize=90)	4	2.99	780.0	1617.0
HeadDetector (minHeadSize=90)	8	2.76	978.0	1638.0

15.2.2.4 GPU. HumanFace detector performance

The table below shows the performance of HumanFaceDetector on the GPU.

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociatior (minFaceSize=20)	1	16.79	850.0	1938.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	4	15.41	1694.0	2032.0
HumanFaceDetectorBoxesAndAssociatior (minFaceSize=20)	8	15.94	2812.0	2118.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	1	5.53	626.0	1953.0
HumanFaceDetectorBoxesAndAssociatior (minFaceSize=50)	4	3.9	926.0	1954.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	8	3.66	1298.0	2012.0
HumanFaceDetectorBoxesAndAssociatior (minFaceSize=90)	1	5.39	618.0	1939.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	4	2.53	718.0	1956.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociatior (minFaceSize=90)	8	2.29	846.0	1983.0
HumanFaceDetectorBoxes (minFaceSize=20)	1	14.6	842.0	1937.0
HumanFaceDetectorBoxes (minFaceSize=20)	4	13.26	1662.0	1956.0
HumanFaceDetectorBoxes (minFaceSize=20)	8	13.72	2748.0	2022.0
HumanFaceDetectorBoxes (minFaceSize=50)	1	5.14	624.0	1932.0
HumanFaceDetectorBoxes (minFaceSize=50)	4	3.61	914.0	1960.0
HumanFaceDetectorBoxes (minFaceSize=50)	8	3.4	1274.0	1979.0
HumanFaceDetectorBoxes (minFaceSize=90)	1	4.23	618.0	1937.0
HumanFaceDetectorBoxes (minFaceSize=90)	4	2.55	714.0	1955.0
HumanFaceDetectorBoxes (minFaceSize=90)	8	2.18	838.0	1979.0

15.2.2.5 GPU. Estimations performance with batch interface

The table below shows the performance of Estimations on the GPU for estimators that have a batch interface. All these measurements are performed with `minFaceSize=50`.

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
HeadPose	1	1.97	670.0	1699.0
HeadPose	16	1.43	766.0	1785.0
HeadPose	32	1.4	860.0	1882.0
Warper	1	0.09	676.0	1685.0
Warper	16	0.05	772.0	1678.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
Warper	32	0.03	874.0	1673.0
Eyes (RGB, useStatusPlan=0)	1	0.78	670.0	1690.0
Eyes (RGB, useStatusPlan=0)	16	0.13	702.0	1701.0
Eyes (RGB, useStatusPlan=0)	32	0.16	702.0	1713.0
Eyes (RGB, useStatusPlan=1)	1	0.76	670.0	1697.0
Eyes (RGB, useStatusPlan=1)	16	0.13	702.0	1691.0
Eyes (RGB, useStatusPlan=1)	32	0.12	702.0	1701.0
Eyes (INFRA RED, useStatusPlan=0)	1	0.6	586.0	1698.0
Eyes (INFRA RED, useStatusPlan=0)	16	0.11	586.0	1693.0
Eyes (INFRA RED, useStatusPlan=0)	32	0.09	618.0	1697.0
Eyes (INFRA RED, useStatusPlan=1)	1	0.43	586.0	1695.0
Eyes (INFRA RED, useStatusPlan=1)	16	0.1	586.0	1690.0
Eyes (INFRA RED, useStatusPlan=1)	32	0.13	618.0	1696.0
InfraRed	1	0.92	600.0	1658.0
InfraRed	16	0.52	638.0	1697.0
InfraRed	32	0.52	674.0	1702.0
AGS	1	1.99	712.0	1677.0
AGS	16	1.42	712.0	1778.0
AGS	32	1.39	902.0	1864.0
BestShotQuality	1	2.38	712.0	1683.0
BestShotQuality	16	1.44	744.0	1775.0
BestShotQuality	32	1.41	934.0	1864.0
MedicalMask	1	4.95	702.0	2024.0
MedicalMask	16	1.67	798.0	2108.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
MedicalMask	32	1.6	860.0	2203.0
LivenessOneShotRGBEstimator 2XL	1	27.06	1324.0	2063.0
LivenessOneShotRGBEstimator 2XL	8	31.18	2396.0	2226.0
LivenessOneShotRGBEstimator 2XL	16	143.01	4116.0	2236.0
LivenessOneShotRGBEstimator XL	1	12.93	924.0	1871.0
LivenessOneShotRGBEstimator XL	8	9.23	1410.0	1875.0
LivenessOneShotRGBEstimator XL	16	8.75	1778.0	1888.0
Orientation	1	2.6	580.0	1634.0
Orientation	16	0.65	696.0	1636.0
Orientation	32	0.64	824.0	1640.0
FacialHair	1	2.03	712.0	1717.0
FacialHair	16	0.72	712.0	1708.0
FacialHair	32	0.7	896.0	1716.0
CredibilityCheck	1	5.07	712.0	1789.0
CredibilityCheck	16	3.44	1330.0	1772.0
CredibilityCheck	32	3.38	1948.0	1802.0
BlackWhite	1	1.08	712.0	1699.0
BlackWhite	16	0.3	744.0	1698.0
BlackWhite	32	0.29	744.0	1708.0
NaturalLight	1	2.01	744.0	1710.0
NaturalLight	16	0.22	744.0	1702.0
NaturalLight	32	0.21	744.0	1705.0
PortraitStyle	1	2.4	712.0	1649.0
PortraitStyle	16	1.52	712.0	1795.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
PortraitStyle	32	1.48	902.0	1883.0
FishEye	1	1.22	712.0	1686.0
FishEye	16	0.23	744.0	1684.0
FishEye	32	0.21	712.0	1688.0
EyeBrow	1	2.05	712.0	1706.0
EyeBrow	16	0.74	712.0	1707.0
EyeBrow	32	0.7	896.0	1714.0
HumanAttribute	1	3.15	602.0	1722.0
HumanAttribute	16	0.9	704.0	1718.0
HumanAttribute	32	0.64	836.0	1717.0
RedEye	1	1.08	712.0	1684.0
RedEye	16	0.2	712.0	1681.0
RedEye	32	0.18	712.0	1678.0
HeadWear	1	2.32	712.0	1722.0
HeadWear	16	0.41	744.0	1724.0
HeadWear	32	0.27	712.0	1710.0
Background	1	2.42	712.0	1630.0
Background	16	1.54	712.0	1780.0
Background	32	1.45	902.0	1879.0
Mouth	1	1.65	744.0	1716.0
Mouth	16	0.45	744.0	1719.0
Mouth	32	0.42	940.0	1699.0
Attributes (netType=0, precise)	1	3.1	712.0	1784.0
Attributes (netType=0, precise)	16	1.93	1214.0	1761.0
Attributes (netType=0, precise)	32	1.9	1736.0	1755.0
Attributes (netType=1, fast)	1	1.9	744.0	1707.0
Attributes (netType=1, fast)	16	0.5	744.0	1706.0
Attributes (netType=1, fast)	32	0.5	744.0	1711.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
Subjective Quality	1	0.64	670.0	1687.0
Subjective Quality	16	0.14	702.0	1690.0
Subjective Quality	32	0.12	702.0	1688.0
Emotions	1	1.95	712.0	1707.0
Emotions	16	0.74	712.0	1714.0
Emotions	32	0.72	896.0	1712.0
EyesGaze	1	1.17	712.0	1626.0
EyesGaze	16	0.45	712.0	1700.0
EyesGaze	32	0.42	712.0	1706.0
Glasses	1	0.97	712.0	1699.0
Glasses	16	0.17	712.0	1693.0
Glasses	32	0.15	712.0	1695.0
LivenessFlyingFaces	1	3.58	744.0	1713.0
LivenessFlyingFaces	16	1.95	1034.0	1808.0
LivenessFlyingFaces	32	1.91	1220.0	1892.0
DynamicRange	1	1.76	712.0	1625.0
DynamicRange	16	1.46	712.0	1715.0
DynamicRange	32	1.46	902.0	1810.0
Ethnicity	1	1.79	712.0	1714.0
Ethnicity	16	0.72	712.0	1710.0
Ethnicity	32	0.71	896.0	1701.0
DeepFake	1	9.41	758.0	1783.0
DeepFake	16	8.09	2142.0	1893.0
DeepFake	32	8.91	3590.0	1981.0
Fights	1	14.31	928.0	1855.0
NIRLivenessEstimator	1	9.35	616.0	1661.0
NIRLivenessEstimator	16	7.75	714.0	1752.0
NIRLivenessEstimator	32	8.22	842.0	1863.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
LivenessRGBMEstimator	1	7.46	712.0	1696.0
LivenessRGBMEstimator	16	4.25	1466.0	1821.0
LivenessRGBMEstimator	32	4.48	2066.0	1956.0
DepthLivenessEstimator	1	1.87	654.0	1676.0
DepthLivenessEstimator	16	0.46	612.0	1662.0
DepthLivenessEstimator	32	0.39	646.0	1675.0
YUV12toRGB	1	2.61	114.0	229.0
YUV12toRGB	16	2.63	114.0	228.0
YUV12toRGB	32	2.56	114.0	227.0
YUV21toRGB	1	3.16	126.0	248.0
YUV21toRGB	16	3.07	126.0	247.0
YUV21toRGB	32	3.09	126.0	246.0
Rotation	1	0.91	114.0	210.0
FaceOcclusion	1	1.96	744.0	1673.0
FaceOcclusion	16	0.79	968.0	1672.0
FaceOcclusion	32	0.76	1160.0	1678.0
ImageModification	1	4.13	600.0	1703.0
ImageModification	16	1.96	812.0	1783.0
ImageModification	32	2.02	1034.0	1878.0

15.2.2.6 GPU. Estimations performance without batch interface

The table below shows the performance of Estimations on the GPU for estimators that do not have a batch interface. All these measurements are performed with `minFaceSize=50`.

Measurement	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
PPE	3.09	662.0	1676.0
Overlap	0.7	670.0	1665.0

15.2.2.7 GPU. Extractor performance

The table below shows the performance of Extractor on the GPU.

Model	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
58	1	9.57	0.0	1844.0
58	8	6.58	0.0	1835.0
58	16	6.24	0.0	1837.0
59	1	9.66	754.0	1848.0
59	8	6.68	866.0	1834.0
59	16	6.34	976.0	1836.0
60	1	9.67	0.0	1844.0
60	8	6.68	0.0	1842.0
60	16	6.32	0.0	1839.0
62	1	11.05	796.0	1877.0
62	8	7.97	904.0	1875.0
62	16	7.71	1010.0	1878.0
65	1	2.83	0.0	1728.0
65	8	1.42	0.0	1736.0
65	16	1.37	0.0	1752.0
105	1	3.48	785	1664
105	16	0.3	815	1673
106	1	6.28	973	1893
106	16	9.38	1371	1894
107	1	3.41	807	1698
107	16	0.59	911	1696
108	1	2.65	586.0	1675.0
108	8	0.5	586.0	1671.0
108	16	0.35	608.0	1675.0
109	1	6.22	933	1833
109	16	7.83	1261	1833
110	1	3.38	809	1693

Model	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
110	16	0.76	939	1693
112	1	5.6	0.0	1766.0
112	8	4.31	0.0	1775.0
112	16	2.95	0.0	1772.0
113	1	2.65	0.0	1695.0
113	8	0.78	0.0	1680.0
113	16	0.97	0.0	1687.0
115	1	5.65	702.0	1776.0
115	8	4.35	766.0	1777.0
115	16	2.96	828.0	1771.0
116	1	2.6	610.0	1689.0
116	8	0.81	640.0	1683.0
116	16	1.05	670.0	1689.0

15.2.2.8 GPU. CrowdEstimator performance

The table below shows the performance of CrowdEstimator on the GPU.

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
CrowdEstimator (Single, minHeadSize=6)	1	55.64	1534.0	1779.0
CrowdEstimator (Single, minHeadSize=6)	4	58.58	3528.0	1810.0
CrowdEstimator (Single, minHeadSize=6)	8	58.85	3426.0	1839.0
CrowdEstimator (Single, minHeadSize=12)	1	18.75	1008.0	1769.0
CrowdEstimator (Single, minHeadSize=12)	4	17.7	1722.0	1790.0
CrowdEstimator (Single, minHeadSize=12)	8	17.96	1766.0	1815.0

Measurement	Batch Size	Percentile 95 (ms)	GPU Memory (Mb)	RAM Memory (Mb)
CrowdEstimator (TwoNets, minHeadSize=6)	1	64.16	1712.0	1782.0
CrowdEstimator (TwoNets, minHeadSize=6)	4	69.99	4192.0	1830.0
CrowdEstimator (TwoNets, minHeadSize=6)	8	68.9	5006.0	1857.0
CrowdEstimator (TwoNets, minHeadSize=12)	1	22.73	1078.0	1784.0
CrowdEstimator (TwoNets, minHeadSize=12)	4	20.75	1946.0	1802.0
CrowdEstimator (TwoNets, minHeadSize=12)	8	21.14	2194.0	1828.0

> **Note:** GPU Memory values reported as 0.0 indicate that GPU memory consumption measurements are not currently available for those algorithms. These measurements may not be collected on a regular basis for all algorithms.

15.2.3 Rockchip (Ubuntu 24.04 LTS)

The number of threads `auto` means that will be taken the maximum number of available threads. For this mode use the `-1` value for the `numThreads` parameter in the `runtime.conf`. This number of threads is equal to according number of available processor cores. We strongly recommend you to follow this recommendation; otherwise, performance can be significantly reduced. Description of according settings you can find in “Configuration Guide - Runtime settings”.

The performance measurements are presented for device with configurations as below:

Architecture: aarch64 Byte Order: Little Endian CPU(s): 8 On-line CPU(s) list: 0-7 Thread(s) per core: 1 Core(s) per socket: 4 Socket(s): 1 Vendor ID: ARM Model: 0 Model name: Cortex-A55 Stepping: r2p0 CPU max MHz: 1800.0000 CPU min MHz: 408.0000 BogoMIPS: 48.00 Flags: fp asimd evtstrm aes pmull sha1 sha2 crc32 atomics fphp asimdhp cpuid asimdrdm lrcpc dcpop asimddp

The number of threads you can find in tables below.

***Note:** In the case of these tests, `power` and `weak` refer to a Linux command (taskset -c j,k, where `j` and `k` are CPU cores) that explicitly sets the CPU affinity of a process. In simple terms, it tells the system to run the process only on the specified CPU cores. `Power` stands for `taskset -c 4-7` and `weak` stands for `taskset -c 0-3`.

15.2.3.1 Rockchip (power) environment. Detector performance

The table below shows the performance of Detector on the Rockchip (*power*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Detector (minFaceSize=20)	1	1	4199.43	604.0
Detector (minFaceSize=20)	4	1	2408.09	650.0
Detector (minFaceSize=20)	4	8	2110.99	4716.0
Detector (minFaceSize=50)	1	1	540.44	137.0
Detector (minFaceSize=50)	4	1	394.12	167.0
Detector (minFaceSize=50)	4	8	315.08	829.0
Detector (minFaceSize=90)	1	1	169.76	76.0
Detector (minFaceSize=90)	4	1	88.09	106.0
Detector (minFaceSize=90)	4	8	124.26	344.0
Redetect	1	1	3.36	130.0
Redetect	4	1	202.56	136.0
Redetect	4	8	26.96	777.0
Landmarks5Detector	1	1	1.12	140.0
Landmarks5Detector	4	1	0.74	141.0
Landmarks5Detector	4	8	0.62	141.0
Landmarks68Detector	1	1	8.34	140.0
Landmarks68Detector	4	1	5.92	141.0
Landmarks68Detector	4	8	5.5	141.0

15.2.3.2 Rockchip (power) environment. Extractor performance

The table below shows the performance of Extractor on the Rockchip (*power*) environment.

Model	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
62	1	1	2130.74	389.0
62	2	1	2110.69	387.0
62	2	8	2216.14	387.0

15.2.3.3 Rockchip (power) environment. HeadDetector performance

The table below shows the performance of HeadDetector on the Rockchip (*power*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadDetector (minHeadSize=20)	1	1	4202.24	599.0
HeadDetector (minHeadSize=20)	4	1	2310.89	633.0
HeadDetector (minHeadSize=20)	4	8	2169.25	4708.0
HeadDetector (minHeadSize=50)	1	1	532.89	131.0
HeadDetector (minHeadSize=50)	4	1	483.73	158.0
HeadDetector (minHeadSize=50)	4	8	318.0	825.0
HeadDetector (minHeadSize=90)	1	1	155.04	71.0
HeadDetector (minHeadSize=90)	4	1	219.86	98.0
HeadDetector (minHeadSize=90)	4	8	123.74	339.0

15.2.3.4 Rockchip (power) environment. HumanDetector performance

The table below shows the performance of HumanDetector on the Rockchip (*power*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanDetector (resize to 320)	1	1	70.32	56.0
HumanDetector (resize to 320)	2	1	80.67	55.0
HumanDetector (resize to 320)	2	8	83.49	177.0
HumanDetector (resize to 640)	1	1	316.34	89.0
HumanDetector (resize to 640)	2	1	321.24	90.0
HumanDetector (resize to 640)	2	8	352.26	454.0
HumanRedetect	1	1	14.49	88.0
HumanRedetect	2	1	14.38	88.0
HumanRedetect	2	8	14.42	413.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	1	1	4657.1	605.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	4	1	2483.13	652.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	4	8	2483.89	4760.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	1	1	581.11	131.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	4	1	608.9	158.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	4	8	341.92	832.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	1	1	179.63	70.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	4	1	199.67	97.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	4	8	154.09	340.0
HumanFaceDetectorBoxes (minFaceSize=20)	1	1	4830.09	581.0
HumanFaceDetectorBoxes (minFaceSize=20)	4	1	2720.71	629.0
HumanFaceDetectorBoxes (minFaceSize=20)	4	8	2467.61	4602.0
HumanFaceDetectorBoxes (minFaceSize=50)	1	1	599.35	128.0
HumanFaceDetectorBoxes (minFaceSize=50)	4	1	501.08	155.0
HumanFaceDetectorBoxes (minFaceSize=50)	4	8	348.5	806.0
HumanFaceDetectorBoxes (minFaceSize=90)	1	1	172.04	69.0
HumanFaceDetectorBoxes (minFaceSize=90)	4	1	294.96	99.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxes (minFaceSize=90)	4	8	128.67	331.0
HumanWarper	1	1	0.64	51.0
HumanWarper	2	1	0.59	52.0
HumanWarper	2	8	1.02	93.0
HumanWarper	1	1	0.64	86.0
HumanWarper	2	1	0.61	87.0
HumanWarper	2	8	1.01	128.0

15.2.3.5 Rockchip (power) Estimations performance without batch interface

The table below shows the performance of Estimations on the CPU for estimators that do not have a batch interface. All these measurements are performed with `minFaceSize=50`.

Type	CPU threads	Percentile 95 (ms)	RAM Memory (Mb)
PPE	4	38.72	100.0
PPE	1	68.62	100.0
Overlap	4	15.56	139.0
Overlap	1	29.45	140.0

15.2.3.6 Rockchip (power) environment. Estimations performance with batch interface

The table below shows the performance of Estimations on the Rockchip (*power*) environment for estimators that have a batch interface.

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadPose	1	1	0.95	139.0
HeadPose	4	1	300.84	138.0
HeadPose	4	8	13.06	180.0
Warper	1	1	4.62	130.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Warper	4	1	310.52	131.0
Warper	4	8	36.49	139.0
Eyes (RGB, useStatusPlan=0)	1	1	4.84	134.0
Eyes (RGB, useStatusPlan=0)	4	1	503.16	134.0
Eyes (RGB, useStatusPlan=0)	4	8	27.58	136.0
Eyes (RGB, useStatusPlan=1)	1	1	4.74	134.0
Eyes (RGB, useStatusPlan=1)	4	1	2.43	134.0
Eyes (RGB, useStatusPlan=1)	4	8	16.75	136.0
Eyes (INFRA RED, useStatusPlan=0)	1	1	2.61	48.0
Eyes (INFRA RED, useStatusPlan=0)	4	1	1.56	48.0
Eyes (INFRA RED, useStatusPlan=0)	4	8	14.05	54.0
Eyes (INFRA RED, useStatusPlan=1)	1	1	2.59	48.0
Eyes (INFRA RED, useStatusPlan=1)	4	1	101.56	48.0
Eyes (INFRA RED, useStatusPlan=1)	4	8	14.07	54.0
InfraRed	1	1	13.69	51.0
InfraRed	4	1	7.48	51.0
InfraRed	4	8	12.04	71.0
AGS	1	1	0.91	138.0
AGS	4	1	100.65	138.0
AGS	4	8	0.56	180.0
BestShotQuality	1	1	1.97	140.0
BestShotQuality	4	1	201.53	140.0
BestShotQuality	4	8	62.36	182.0
MedicalMask	1	1	37.53	157.0
MedicalMask	4	1	20.33	158.0
MedicalMask	4	8	43.37	200.0
Orientation	1	1	47.56	37.0
Orientation	4	1	227.6	42.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Orientation	4	8	62.78	85.0
FacialHair	1	1	123.39	150.0
FacialHair	4	1	59.53	150.0
FacialHair	4	8	78.8	150.0
CredibilityCheck	1	1	1148.36	224.0
CredibilityCheck	4	1	584.05	224.0
CredibilityCheck	4	8	622.93	224.0
BlackWhite	1	1	7.71	136.0
BlackWhite	4	1	4.19	137.0
BlackWhite	4	8	4.33	139.0
NaturalLight	1	1	15.09	140.0
NaturalLight	4	1	8.83	141.0
NaturalLight	4	8	19.91	140.0
PortraitStyle	1	1	6.62	138.0
PortraitStyle	4	1	4.26	138.0
PortraitStyle	4	8	4.36	180.0
FishEye	1	1	16.56	141.0
FishEye	4	1	10.31	143.0
FishEye	4	8	10.09	143.0
EyeBrow	1	1	119.65	150.0
EyeBrow	4	1	158.65	150.0
EyeBrow	4	8	91.5	149.0
HumanAttribute	1	1	89.08	59.0
HumanAttribute	4	1	47.84	59.0
HumanAttribute	4	8	50.29	78.0
RedEye	1	1	17.98	135.0
RedEye	4	1	9.17	135.0
RedEye	4	8	34.74	136.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadWear	1	1	26.77	150.0
HeadWear	4	1	15.22	150.0
HeadWear	4	8	37.24	150.0
Background	1	1	6.75	138.0
Background	4	1	4.13	138.0
Background	4	8	4.09	180.0
Mouth	1	1	51.01	141.0
Mouth	4	1	28.2	142.0
Mouth	4	8	44.23	141.0
Attributes	1	1	524.81	182.0
Attributes	4	1	252.39	182.0
Attributes	4	8	308.9	274.0
Subjective Quality	1	1	6.35	133.0
Subjective Quality	4	1	3.37	133.0
Subjective Quality	4	8	3.73	133.0
Emotions	1	1	118.09	149.0
Emotions	4	1	158.05	149.0
Emotions	4	8	102.62	149.0
EyesGaze	1	1	15.69	135.0
EyesGaze	4	1	8.66	136.0
EyesGaze	4	8	9.25	136.0
Glasses	1	1	6.32	134.0
Glasses	4	1	3.91	134.0
Glasses	4	8	3.57	134.0
LivenessFlyingFaces	1	1	86.33	156.0
LivenessFlyingFaces	4	1	142.59	183.0
LivenessFlyingFaces	4	8	67.67	220.0
DynamicRange	1	1	0.48	135.0

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
DynamicRange	4	1	92.53	136.0
DynamicRange	4	8	53.96	178.0
Ethnicity	1	1	113.51	149.0
Ethnicity	4	1	159.18	149.0
Ethnicity	4	8	80.65	149.0
LivenessRGBMEstimator	1	1	210.64	142.0
LivenessRGBMEstimator	4	1	109.86	141.0
LivenessRGBMEstimator	4	8	131.9	406.0
YUV12toRGB	1	1	1.73	28.0
YUV12toRGB	4	1	1.67	28.0
YUV12toRGB	4	8	1.66	28.0
YUV21toRGB	1	1	2.35	29.0
YUV21toRGB	4	1	1.75	29.0
YUV21toRGB	4	8	2.31	29.0
Rotation	1	1	17.49	36.0
Rotation	4	1	22.34	36.0
FaceOcclusion	1	1	49.82	133.0
FaceOcclusion	4	1	28.3	134.0
FaceOcclusion	4	8	43.68	134.0
LivenessOneShotRGBEstimator M	1	1	653.29	192.0
LivenessOneShotRGBEstimator M	4	1	597.94	193.0
LivenessOneShotRGBEstimator M	4	8	623.12	289.0
LivenessOneShotRGBEstimator S	1	1	195.13	154.0
LivenessOneShotRGBEstimator S	4	1	111.6	151.0
LivenessOneShotRGBEstimator S	4	8	113.08	263.0

15.2.3.7 Rockchip (weak) environment. Detector performance

The table below shows the performance of Detector on the Rockchip (*weak*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Detector (minFaceSize=20)	1	1	13355.6	604.0
Detector (minFaceSize=20)	4	1	6902.77	650.0
Detector (minFaceSize=20)	4	8	7023.7	4715.0
Detector (minFaceSize=50)	1	1	1908.48	136.0
Detector (minFaceSize=50)	4	1	1180.17	163.0
Detector (minFaceSize=50)	4	8	1052.15	834.0
Detector (minFaceSize=90)	1	1	569.72	76.0
Detector (minFaceSize=90)	4	1	481.89	102.0
Detector (minFaceSize=90)	4	8	365.46	351.0
Redetect	1	1	12.56	130.0
Redetect	4	1	409.6	134.0
Redetect	4	8	31.97	788.0
Landmarks5Detector	1	1	4.35	140.0
Landmarks5Detector	4	1	203.27	141.0
Landmarks5Detector	4	8	39.96	141.0
Landmarks68Detector	1	1	35.99	140.0
Landmarks68Detector	4	1	223.31	141.0
Landmarks68Detector	4	8	45.92	141.0

15.2.3.8 Rockchip (weak) environment. Extractor performance

The table below shows the performance of Extractor on the Rockchip (*weak*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Extractor	1	1	8613.25	389.0
Extractor	4	1	4397.11	387.0

15.2.3.9 Rockchip (weak) environment. HeadDetector performance

The table below shows the performance of HeadDetector on the Rockchip (*weak*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadDetector (minHeadSize=20)	1	1	13130.7	599.0
HeadDetector (minHeadSize=20)	4	1	7099.49	632.0
HeadDetector (minHeadSize=20)	4	8	7089.21	4517.0
HeadDetector (minHeadSize=50)	1	1	1854.69	131.0
HeadDetector (minHeadSize=50)	4	1	1357.85	158.0
HeadDetector (minHeadSize=50)	4	8	1042.11	825.0
HeadDetector (minHeadSize=90)	1	1	611.98	71.0
HeadDetector (minHeadSize=90)	4	1	607.73	98.0
HeadDetector (minHeadSize=90)	4	8	361.1	338.0

15.2.3.10 Rockchip (weak) environment. HumanDetector performance

The table below shows the performance of HumanDetector on the Rockchip (*weak*) environment.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanDetector (resize to 320)	1	1	299.54	57.0
HumanDetector (resize to 320)	4	1	269.02	78.0
HumanDetector (resize to 320)	4	8	191.12	190.0
HumanDetector (resize to 640)	1	1	1243.92	90.0
HumanDetector (resize to 640)	4	1	805.75	115.0
HumanDetector (resize to 640)	4	8	696.83	470.0
HumanRedetect	1	1	55.02	88.0
HumanRedetect	4	1	231.87	104.0
HumanRedetect	4	8	55.61	428.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	1	1	14804.5	605.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	4	1	7646.54	653.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanFaceDetectorBoxesAndAssociations (minFaceSize=20)	4	8	7754.83	4572.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	1	1	2058.23	133.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	4	1	1368.82	161.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=50)	4	8	1118.33	834.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	1	1	640.82	72.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	4	1	537.05	99.0
HumanFaceDetectorBoxesAndAssociations (minFaceSize=90)	4	8	378.61	349.0
HumanFaceDetectorBoxes (minFaceSize=20)	1	1	14719.2	584.0
HumanFaceDetectorBoxes (minFaceSize=20)	4	1	7480.31	631.0
HumanFaceDetectorBoxes (minFaceSize=20)	4	8	7674.93	4417.0
HumanFaceDetectorBoxes (minFaceSize=50)	1	1	2040.64	130.0
HumanFaceDetectorBoxes (minFaceSize=50)	4	1	1263.58	157.0
HumanFaceDetectorBoxes (minFaceSize=50)	4	8	1118.03	808.0
HumanFaceDetectorBoxes (minFaceSize=90)	1	1	660.46	71.0
HumanFaceDetectorBoxes (minFaceSize=90)	4	1	642.33	99.0
HumanFaceDetectorBoxes (minFaceSize=90)	4	8	395.33	334.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HumanWarper	1	1	2.34	54.0
HumanWarper	4	1	167.79	54.0
HumanWarper	4	8	25.04	95.0
HumanWarper	1	1	2.37	88.0
HumanWarper	4	1	288.84	89.0
HumanWarper	4	8	25.03	130.0

15.2.3.11 Rockchip (weak) Estimations performance without batch interface

The table below shows the performance of Estimations on the CPU for estimators that do not have a batch interface. All these measurements are performed with `minFaceSize=50`.

Type	CPU threads	Percentile 95 (ms)	RAM Memory (Mb)
PPE	4	88.89	100.0
PPE	1	277.88	100.0
Overlap	4	32.46	139.0
Overlap	1	110.82	140.0

15.2.3.12 Rockchip (weak) environment. Estimations performance with batch interface

The table below shows the performance of Estimations on the Rockchip (*weak*) environment for estimators that have a batch interface.

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadPose	1	1	3.4	139.0
HeadPose	4	1	202.86	138.0
HeadPose	4	8	51.74	179.0
Warper	1	1	18.21	130.0
Warper	4	1	223.48	130.0
Warper	4	8	44.46	137.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Eyes (RGB, useStatusPlan=0)	1	1	20.9	134.0
Eyes (RGB, useStatusPlan=0)	4	1	313.57	134.0
Eyes (RGB, useStatusPlan=0)	4	8	47.09	135.0
Eyes (RGB, useStatusPlan=1)	1	1	23.36	134.0
Eyes (RGB, useStatusPlan=1)	4	1	312.58	134.0
Eyes (RGB, useStatusPlan=1)	4	8	35.24	136.0
Eyes (INFRA RED, useStatusPlan=0)	1	1	10.18	48.0
Eyes (INFRA RED, useStatusPlan=0)	4	1	206.16	48.0
Eyes (INFRA RED, useStatusPlan=0)	4	8	18.86	50.0
Eyes (INFRA RED, useStatusPlan=1)	1	1	10.54	48.0
Eyes (INFRA RED, useStatusPlan=1)	4	1	469.17	48.0
Eyes (INFRA RED, useStatusPlan=1)	4	8	31.17	50.0
InfraRed	1	1	53.7	51.0
InfraRed	4	1	27.72	50.0
InfraRed	4	8	34.38	71.0
AGS	1	1	3.52	138.0
AGS	4	1	300.8	138.0
AGS	4	8	26.89	180.0
BestShotQuality	1	1	8.66	140.0
BestShotQuality	4	1	406.05	140.0
BestShotQuality	4	8	28.78	182.0
MedicalMask	1	1	191.77	157.0
MedicalMask	4	1	397.13	159.0
MedicalMask	4	8	104.14	200.0
Orientation	1	1	199.5	37.0
Orientation	4	1	308.51	39.0
Orientation	4	8	164.6	86.0
FacialHair	1	1	476.09	150.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
FacialHair	4	1	439.41	150.0
FacialHair	4	8	265.04	149.0
CredibilityCheck	1	1	4419.75	224.0
CredibilityCheck	4	1	2402.02	224.0
CredibilityCheck	4	8	2444.64	224.0
BlackWhite	1	1	31.5	136.0
BlackWhite	4	1	16.85	137.0
BlackWhite	4	8	17.49	138.0
NaturalLight	1	1	73.54	140.0
NaturalLight	4	1	239.47	141.0
NaturalLight	4	8	55.71	141.0
PortraitStyle	1	1	28.65	138.0
PortraitStyle	4	1	316.93	139.0
PortraitStyle	4	8	52.92	180.0
FishEye	1	1	67.25	141.0
FishEye	4	1	238.72	143.0
FishEye	4	8	59.63	143.0
EyeBrow	1	1	454.11	150.0
EyeBrow	4	1	442.95	150.0
EyeBrow	4	8	297.56	150.0
HumanAttribute	1	1	380.23	59.0
HumanAttribute	4	1	196.17	59.0
HumanAttribute	4	8	209.11	75.0
RedEye	1	1	70.76	135.0
RedEye	4	1	239.34	135.0
RedEye	4	8	46.07	135.0
HeadWear	1	1	138.35	150.0
HeadWear	4	1	216.67	150.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
HeadWear	4	8	79.63	149.0
Background	1	1	28.39	138.0
Background	4	1	417.31	139.0
Background	4	8	27.68	180.0
Mouth	1	1	201.22	141.0
Mouth	4	1	306.79	141.0
Mouth	4	8	139.64	141.0
Attributes	1	1	2546.66	182.0
Attributes	4	1	1397.4	182.0
Attributes	4	8	1307.17	275.0
Subjective Quality	1	1	23.12	133.0
Subjective Quality	4	1	313.21	133.0
Subjective Quality	4	8	37.31	133.0
Emotions	1	1	479.49	149.0
Emotions	4	1	626.0	149.0
Emotions	4	8	284.64	149.0
EyesGaze	1	1	67.54	135.0
EyesGaze	4	1	35.39	136.0
EyesGaze	4	8	33.71	136.0
Glasses	1	1	26.46	133.0
Glasses	4	1	142.21	134.0
Glasses	4	8	38.64	134.0
LivenessFlyingFaces	1	1	316.23	156.0
LivenessFlyingFaces	4	1	467.64	183.0
LivenessFlyingFaces	4	8	197.72	220.0
DynamicRange	1	1	1.55	135.0
DynamicRange	4	1	201.92	136.0
DynamicRange	4	8	27.13	178.0

Type	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Ethnicity	1	1	459.94	149.0
Ethnicity	4	1	428.69	149.0
Ethnicity	4	8	346.07	150.0
LivenessRGBMEstimator	1	1	730.75	142.0
LivenessRGBMEstimator	4	1	393.06	141.0
LivenessRGBMEstimator	4	8	413.06	406.0
YUV12toRGB	1	1	3.68	28.0
YUV12toRGB	4	1	3.49	28.0
YUV12toRGB	4	8	3.6	28.0
YUV21toRGB	1	1	3.76	29.0
YUV21toRGB	4	1	3.7	29.0
YUV21toRGB	4	8	3.73	29.0
Rotation	1	1	54.05	36.0
Rotation	4	1	54.59	36.0
FaceOcclusion	1	1	196.6	133.0
FaceOcclusion	4	1	393.33	134.0
FaceOcclusion	4	8	132.38	133.0
LivenessOneShotRGBEstimator M	1	1	2005.85	192.0
LivenessOneShotRGBEstimator M	4	1	1782.6	193.0
LivenessOneShotRGBEstimator M	4	8	1816.24	289.0
LivenessOneShotRGBEstimator S	1	1	727.97	154.0
LivenessOneShotRGBEstimator S	4	1	601.24	151.0
LivenessOneShotRGBEstimator S	4	8	428.65	263.0

15.3 Runtime performance for embedded environment

Face detection performance depends on input image parameters, including resolution, bit depth, and the size of the detected face.

Input data characteristics:

- Image resolution: 640x480px
- Image format: 24 BPP RGB

The results for both minimum and optimal batch sizes are presented in the tables below, with all intermediate and non-optimal values omitted.

Face detection is performed using the FaceDetV3 neural network.

15.4 Descriptor size

Table below shows size of serialized face descriptors to estimate memory requirements.

Table 109: "Descriptor size"

Face descriptor version	Data size (bytes)	Metadata size (bytes)	Total size
CNN 56	512	8	520
CNN 57	512	8	520
CNN 58	512	8	520
CNN 59	512	8	520
CNN 60	512	8	520
CNN 62	512	8	520
CNN 65	512	8	520

Table below shows size of serialized human descriptors to estimate memory requirements. Human descriptors are used only for reidentification tasks.

Table 110: "Human descriptor size (used only for reidentification tasks)"

Human descriptor version	Data size (bytes)	Metadata size (bytes)	Total size
CNN 102 (deprecated)	2048	8	2056
CNN 103 (deprecated)	2048	8	2056
CNN 104 (deprecated)	2048	8	2056
CNN 105 (deprecated)	512	8	520
CNN 106 (deprecated)	512	8	520
CNN 107 (deprecated)	512	8	520

Human descriptor version	Data size (bytes)	Metadata size (bytes)	Total size
CNN 108	512	8	520
CNN 109 (deprecated)	512	8	520
CNN 110 (deprecated)	512	8	520
CNN 112	512	8	520
CNN 113	512	8	520

Metadata includes signature and version information that may be omitted during serialization if the *NoSignature* flag is specified.

When estimating individual descriptor size in memory or serialization storage requirements with default options, consider using values from the “Total size” column.

When estimating memory requirements for descriptor batches, use values from the “Data size” column instead, since a descriptor batch does not duplicate metadata per descriptor and thus is more memory-efficient.

> These numbers are for approximate computation only, since they do not include overhead like memory alignment for accelerated SIMD processing and the like.

16 Appendix B. Glossary

Table 111: Glossary

Term	Description
Host memory	Computer system RAM
Device memory	On-board RAM of GPU or NPU card
Memory transfer	Operation that copies memory from host to device or vice-versa

16.1 Descriptor

A set of features meant to describe a real-world object (e.g., a person’s face). Computed by means of computer vision algorithms, such features are typically matched to each other to determine the similarity of represented objects.

16.2 Cooperative Photoshooting and Recognition

A procedure of taking person face photograph characterized by person awareness of the matter and his/her will to assist.

Typical highlights:

- Close to frontal head pose;
- Neutral facial expression;
- No occlusions (i.e., hair, hats, non-transparent eyewear, hands, other objects obscuring the face);
- No extreme lighting conditions (i.e., reasonable illuminance, no direct sunlight);
- Steady and well-tuned optics (i.e., no motion blur, depth of field, digital post-processing except noise cancellation).

Cooperative photoshooting is opposite to the so-called "in the wild" photoshooting, which is also called non-cooperative shooting (or recognition).

16.3 Matching

The process of descriptors comparison. Matching is usually implemented as a distance function applied to the feature sets and distances comparison later on. The smaller the distance, the closer are descriptors, hence, the more similar are the objects.

For convenience, helper functions exist to convert distance to a normalized similarity score, where 100% means completely identical, and 0% means completely different.

17 Appendix C. FAQ

Q: This document contains high-level descriptions and no code examples nor reference. Where can one find them?

A: The complete type and function reference are provided as an interactive web-based documentation; see the *doc/fsdk/index.html* inside the LUNA SDK package. The examples are located in the */examples* folder and "ExamplesGuide.pdf" is located in /doc folder of LUNA SDK package.

Q: Does FaceEngine support multicore / multiprocessor systems?

A: Yes, all internal algorithm implementations are multithreaded by design and take advantage of multi-core systems. The number of threads may be controlled via the configuration file; see configuration manual "ConfigurationGuide.pdf" or comments in the configuration file for details.

Q: What is the state of GPU support?

A: As of version 2.7 the GPU support is implemented for face detection and descriptor extraction algorithms. Starting from version 2.9 GPU implementations are considered stable.

Q: What speedup may be expected from GPUs?

A: Typically GPUs allow accelerating algorithms by the factor of 2-4 times depending on microprocessor architecture and input data.

Q: Are there any official bindings/wrappers for other languages (C#, Java)?

A: No, such bindings are not provided. FaceEngine officially implements C++ API only, bindings to other languages should be created by users themselves. There are tools to automate this process, like, e.g., SWIG.

Q: Does FaceEngine support DBMS systems?

A: No, FaceEngine implements just computer vision algorithms. Users should implement DBMS communication themselves using serialization methods described in section "Serializable object interface" of chapter "Core concepts" and section "Archive interface" of chapter "Core facility".

Q: What image formats does FaceEngine support?

A: FaceEngine does not implement image format encoding functions. If such functions are required, one should use a third-party library, e.g., FreeImage.

FaceEngine functions typically expect image data in the form of uncompressed unencoded pixel data (RGB color 24 bits per pixel or grayscale 8 bits per pixel).

FaceEngine implements convenience functions like RGB -> grayscale and RGB<-> BGR color conversions. The rationale of this design is explained in section "Image type" of chapter "Core concepts".

18 Appendix D. Known issues

18.1 Overall known issues

18.1.1 Warnings during the compilation of user code that utilizes the SDK libraries

For example:

```
warning: 'fsdk::IQualityEstimator' has virtual functions but non-virtual
    destructor [-Wnon-virtual-dtor]
        struct IQualityEstimator : IRefCounted {
```

This is a normal and expected behavior. For details, see Core Concepts - Reference Counted Interface.

18.1.2 Premature end of JPEG file

Sometimes you can meet such a log:

```
[Error] [Image] FreeImage error: format=1, msg=Premature end of JPEG file.
```

This issue occurs if your JPEG file was not previously recorded or saved properly. You can find more information on this error on the Internet. Fortunately, this error is not fatal and you can continue working with the image and get valid detection, landmarks and warped image. You can also try to re-save this image.

18.1.3 SDK stuck when run sdk algorithm in separate process after root FaceEngine object initialized

For example:

```
void simpleDetect(const fsdk::Image& image, const fsdk::IDetectorPtr&
    faceDetector) {
     fsdk::ResultValue<fsdk::FSDKError, fsdk::Face> result = faceDetector->
         detectOne(
                image,
                image.getRect(),
                fsdk::DetectionType::DT_BBOX
     );
}

int main()
{
```

```
    auto resFaceEngine = fsdk::createFaceEngine("./data");
    fsdk::IFaceEnginePtr faceEngine = resFaceEngine.getValue();

    fsdk::ILicense* license = faceEngine->getLicense();
    fsdk::activateLicense(license, "./data/license.conf");

    fsdk::Image image;
    const string imagePath {"image_720.jpg"};
    image.load(imagePath.c_str(), fsdk::Format::R8G8B8);

    auto detRes = faceEngine->createDetector(fsdk::FACE_DET_V3);
    fsdk::IDetectorPtr faceDetector = detRes.getValue();

    // Run detection in separate process
    pid_t ch_pid = fork();
    if (ch_pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    } else if (ch_pid > 0) {
        cout << "spawn child with pid - " << ch_pid << endl;
    } else {
        simpleDetect(image, faceDetector);
    }

    pid_t child_pid;
    while ((child_pid = wait(nullptr)) > 0)
        cout << "child " << child_pid << " terminated" << endl;

    return 0;
}
```

Cause deadlock. This behaviour observed since sdk version 5.4 and above. The problem can be solved if you make all forks before creating the FaceEngine object. More reading in Best practices

18.1.4 Undefined behaviour with multithreaded usage of the FaceEngine and algorithms

Creation and destroying Luna SDK algorithms from the different threads is prohibited due to internal implementation restrictions. In such case undefined behaviour is possible - segmentation faults or invalid results. More reading in Best practices

18.1.5 Floating point exceptions when working with images that have GPU memory residence

If you're getting floating point exceptions when using images with GPU memory residence please make sure that Luna SDK runtime has been initialised with at least 2 worker threads. For more info about runtime configuration please refer to Runtime settings chapter in ConfigurationGuide handbook.

18.1.6 Coordinate differences for batched detections

It is possible to obtain some small differences in detected image boxes and landmarks for different placement of images within batches, when the sizes of different images are close to each other. This note is correct for all detector types, including face detectors, human detectors, face+human detectors etc.

18.2 CentOS 8 known issues

18.2.1 Archive unpacking

We have detected such behavior on CentOS 8.

```
unzip *.zip;
error: invalid zip file with overlapped components (possible zip bomb)
```

while unpacking archives. The bug is caused by `unzip-6.0-45.el8` package. We recommend to downgrade it:

```
rpm -q unzip-6.0-45.el8 && yum remove unzip && yum install unzip-6.0-44.el8
```

Possible content of `test.xcent`:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/
    DTDs/PropertyList-1.0.dtd">

<plist version="1.0">

<dict>

    <key>com.apple.security.get-task-allow</key>

    <true/>

</dict>

</plist>
```

18.3 Astra Linux known issues

18.3.1 Startup error

> Applies to LUNA SDK v.5.26.0 and earlier.

This section provides a step-by-step guide to resolving the issue with the **execstack** tool, using the FingerprintViewer application as an example.

The FingerprintViewer application has compatibility issues on Astra Linux because some LUNA SDK shared libraries have the executable stack flag. This flag causes issues on secure operating systems like Astra Linux, which do not allow executable stacks by default.

To fix this, you can remove the executable stack flag from the affected libraries using the execstack tool:

1. Install execstack.

Astra Linux does not include execstack in its standard repositories. To obtain it, you can download the *.deb* package from Debian's official repository.

1.1. Download the package:

```
wget http://ftp.ru.debian.org/debian/pool/main/p/prelink/execstack_0
    .0.20131005-1+b10_amd64.deb
```

Example output:

```
--2025-03-03 11:09:39--  http://ftp.ru.debian.org/debian/pool/main/p/prelink
    /execstack_0.0.20131005-1+b10_amd64.deb
Resolving ftp.ru.debian.org (ftp.ru.debian.org)... 85.143.112.112
Connecting to ftp.ru.debian.org (ftp.ru.debian.org)|85.143.112.112|:80...
    connected.
HTTP request sent, awaiting response... 200 OK
Length: 88380 (86K) [application/octet-stream]
Saving to: ‘execstack_0.0.20131005-1+b10_amd64.’deb

execstack_0.0.20131005-1+b10_amd64.deb
    100%[=============================================================>]
      86.31K  --.-KB/s    in 0.003s

2025-03-03 11:09:39 (28.6 MB/s) - ‘execstack_0.0.20131005-1+b10_amd64.’deb
    saved [88380/88380]
```

1.2. Install the package:

```
sudo apt install ./execstack_0.0.20131005-1+b10_amd64.deb
```

Example output:

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'execstack' instead of './execstack_0.0.20131005-1+b10_amd64
    .deb'
execstack is already the newest version (0.0.20131005-1+b10).
Updated 0 packages, installed 0 new packages, marked 0 for removal, and left
     0 unchanged.
```

2. Identify libraries with executable stack flags.

2.1. Navigate to the directory containing the LUNA SDK shared libraries being used. For example:

```
cd ~/luna-sdk_astra_se_rel_v.X.X.X/lib/gcc4/x64
```

2.2. Run the `execstack` command to check for libraries with the executable stack flag:

```
execstack *
```

Libraries marked with `X` have the executable stack flag set and require modification:

```
- libcublasLt.so
- libcublasLt.so.11
- libcublasLt.so.11.6.5.2
- libcublas.so
- libcublas.so.11
- libcublas.so.11.6.5.2
- libcudnn_cnn_infer.so
- libcudnn_cnn_infer.so.8
- libcudnn_cnn_infer.so.8.9.0
- libcudnn_ops_infer.so
- libcudnn_ops_infer.so.8
- libcudnn_ops_infer.so.8.9.0
- libcudnn.so
- libcudnn.so.8
- libcudnn.so.8.9.0
X libFaceEngineSDK.so
X libFaceEngineSDK.so.5
X libFaceEngineSDK.so.5.23
X libflower.so
- libMatchingKernel.so
- libMatchingKernel.so.0
- libMatchingKernel.so.0.0
X libTrackEngineSDK.so
X libTrackEngineSDK.so.0
X libTrackEngineSDK.so.0.0
X libvlTracker.so
```

3. Remove executable stack flags.

For each library marked with X, use the execstack tool to clear the executable stack flag:

```
execstack -c <library_name>
```

Example commands:

```
execstack -c libFaceEngineSDK.so
execstack -c libFaceEngineSDK.so.5
execstack -c libFaceEngineSDK.so.5.23
execstack -c libflower.so
execstack -c libTrackEngineSDK.so
execstack -c libTrackEngineSDK.so.0
execstack -c libTrackEngineSDK.so.0.0
execstack -c libvlTracker.so
```

4. Verify changes.

After clearing the flags, verify that all shared libraries are free of the executable stack flag:

```
execstack *
```

Expected output after fixing:

```
- libcublasLt.so
- libcublasLt.so.11
- libcublasLt.so.11.6.5.2
- libcublas.so
- libcublas.so.11
- libcublas.so.11.6.5.2
- libcudnn_cnn_infer.so
- libcudnn_cnn_infer.so.8
- libcudnn_cnn_infer.so.8.9.0
- libcudnn_ops_infer.so
- libcudnn_ops_infer.so.8
- libcudnn_ops_infer.so.8.9.0
- libcudnn.so
- libcudnn.so.8
- libcudnn.so.8.9.0
- libFaceEngineSDK.so
- libFaceEngineSDK.so.5
- libFaceEngineSDK.so.5.26
- libflower.so
- libMatchingKernel.so
- libMatchingKernel.so.0
- libMatchingKernel.so.0.0
- libnppc.so
- libnppc.so.11
- libnppc.so.11.4.0.110
- libnppicc.so
- libnppicc.so.11
- libnppicc.so.11.4.0.110
- libnppidei.so
- libnppidei.so.11
- libnppidei.so.11.4.0.110
- libTrackEngineSDK.so
- libTrackEngineSDK.so.0
- libTrackEngineSDK.so.0.0
- libvlTracker.so
```

All libraries should now display a `-` instead of `X`, indicating that the executable stack flag has been successfully removed.

5. Test FingerprintViewer.

After completing the above steps, launch the FingerprintViewer application. If the issue was caused by the executable stack flag, the application should now start without errors.