



VisionLabs
MACHINES CAN SEE

VisionLabs FaceEngine Handbook

written for LUNA SDK Mobile iOS version 5.33.0

Contents

Introduction	6
1 Core Concepts	7
1.1 SDK workflow	7
1.1.1 Object lifetime	7
1.1.2 Threading	8
1.1.3 Detailed constraints	9
1.2 Common Interfaces and Types	10
1.2.1 Reference Counted Interface	10
1.2.2 Automatic reference counting	11
1.2.2.1 Referencing - without acquiring ownership of object lifetime	11
1.2.2.2 Acquiring - own object lifetime	12
1.2.3 Serializable object interface	13
1.2.4 Auxiliary types	13
1.2.4.1 Image type	13
1.3 Beta Mode	14
2 FaceEngine Structure Overview	15
3 Core Facility	16
3.1 Common Interfaces	16
3.1.1 Face Engine Object	16
3.1.2 Settings Provider	16
3.2 Helper interfaces	16
3.2.1 Archive interface	16
3.3 Data Paths	17
3.3.1 Model Data	17
3.3.2 Configuration Data	17
4 Detection facility	18
4.1 Overview	18
4.2 Detection structure	18
4.3 Face Detection	18
4.3.1 Image coordinate system	18
4.3.2 Face detection	19
4.3.3 Redetect method	19
4.3.4 Face Alignment	19
4.3.4.1 Five landmarks	19

5	Image Warping	20
6	Parameter Estimation Facility	22
6.1	Overview	22
6.2	Best shot selection functionality	22
6.2.1	BestShotQuality Estimation	22
6.2.2	Image Quality Estimation	25
6.2.3	Image Subjective Quality Estimation	32
6.3	Face features extraction functionality	38
6.3.1	Eyes Estimation	38
6.4	Head Pose Estimation	40
6.5	Approximate Garbage Score Estimation (AGS)	42
6.6	Glasses Estimation	43
6.7	Liveness check functionality	44
6.8	LivenessOneShotRGB Estimation	44
6.9	Face Occlusion Estimation Functionality	51
6.10	Medical Mask Estimation Functionality	54
6.10.1	MedicalMaskEstimator thresholds	55
6.10.2	MedicalMask enumeration	55
6.10.3	MedicalMaskEstimation structure	56
6.10.4	MedicalMaskExtended enumeration	57
6.10.5	MedicalMaskEstimationExtended structure	57
6.10.6	Filtration parameters	58
7	Descriptor processing facility	60
7.1	Overview	60
7.1.1	Person Identification Task	60
7.2	Descriptor	60
7.2.1	Descriptor Versions	61
7.2.2	Descriptor Batch	61
7.2.3	Descriptor Extraction	62
7.2.4	Descriptor Matching	63
8	System Requirements	65
8.1	IOS installations	65
9	Hardware requirements	65
9.1	Mobile installations	65
9.1.1	CPU requirements	66
9.1.2	Memory requirements	66
9.1.3	Number of threads on mobile devices	66

10 Best practices	67
10.1 Thread pools	67
10.2 Estimator creation and inference	67
11 Device-specific constraints	68
11.1 Image constraints	68
12 Collecting information for Technical Support	69
12.1 Contact Technical Support	69
12.2 Specific error	69
12.3 Non-specific error	70
12.4 Unexpected Result	70
13 Useful tools	72
13.1 Performance testing	72
13.1.1 Key concepts in performance testing	72
13.2 Metrics for performance analysis	72
13.2.1 Common metrics	72
13.2.1.1 Practical use	73
13.3 Performance test parameters	73
13.3.1 Test-specific parameters	73
13.3.2 Batch and sensor parameters	74
13.3.3 iOS-specific parameters	74
13.3.4 Stopping condition parameters	74
13.3.5 Recommendations for parameter selection	75
13.4 Stopping conditions	75
13.4.1 Normal stopping conditions	75
13.4.2 Emergency stopping conditions	76
13.4.2.1 Configuration of emergency stop conditions	76
13.4.3 Special cases	76
13.5 Example console report	77
13.5.1 Structure of the first table	77
13.5.2 Column contents	78
13.5.3 Additional metrics	78
13.5.4 Zero and last iterations	78
13.5.5 Color coding	78
13.5.6 Reasons for stopping	78
13.5.7 Operational vs. final statistics	78
13.6 Performance test challenges	79
13.6.1 Measurement range limitations	79

13.6.2	High-frequency noise	79
13.6.3	Low-frequency noise	79
13.6.4	Test execution duration	79
13.6.5	Artificial constraints efficiency	80
13.6.6	Launch recommendations	80
13.7	Potential improvements	80
13.8	Practical recommendations	80
14	Appendix A. Specifications	81
14.1	Runtime performance for mobile environment	81
14.1.1	IOS	81
14.1.1.1	Matcher performance	81
14.1.1.2	Extractor performance	81
14.1.1.3	Detector performance	82
14.1.1.4	Estimations performance with batch interface	82
14.2	Descriptor size	84
14.3	Feature matrix	85
15	Appendix B. Glossary	86
15.1	Descriptor	86
15.2	Cooperative Photoshooting and Recognition	86
15.3	Matching	86

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.

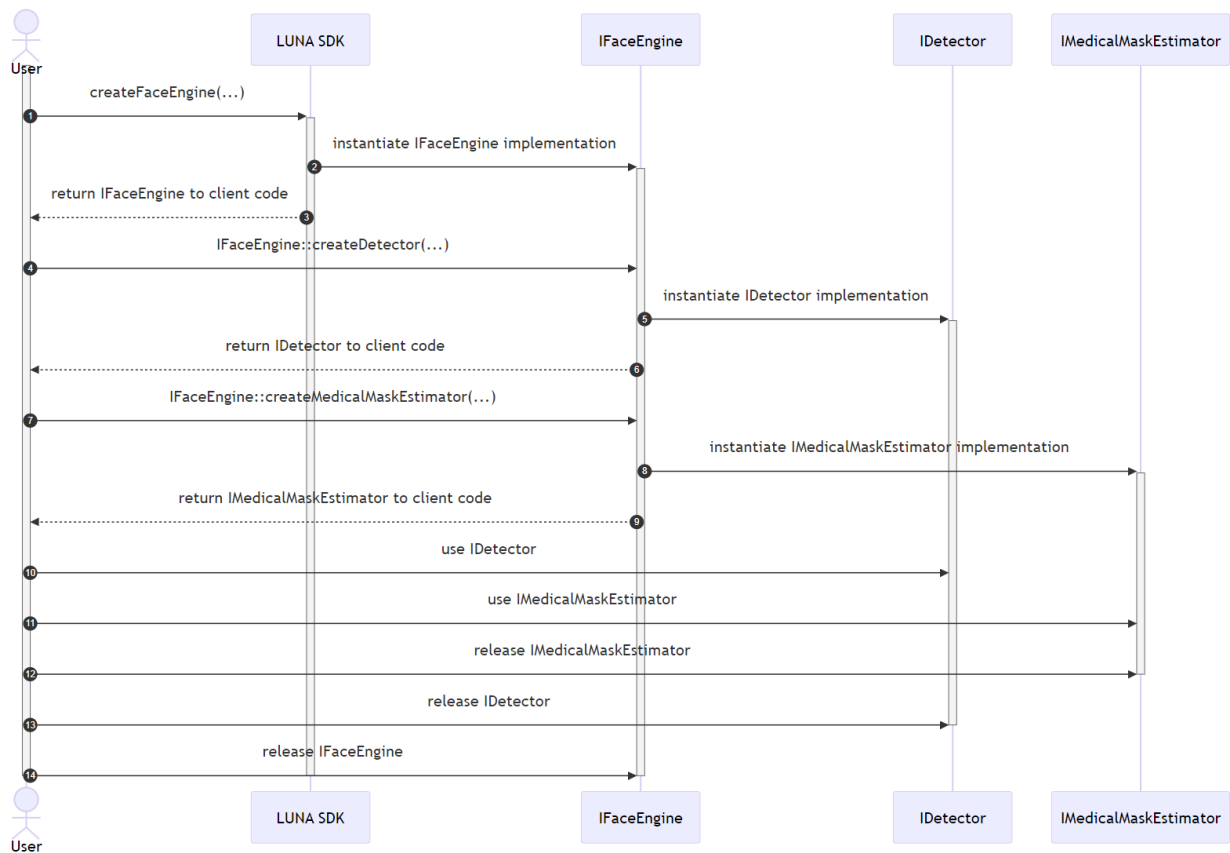
The opening chapter of this book is called “Core concepts”. It will tell you about memory management techniques, object creation and destruction strategies that are widely used across the entire FaceEngine. The following chapters catch up telling how higher level FaceEngine components are created from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

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:

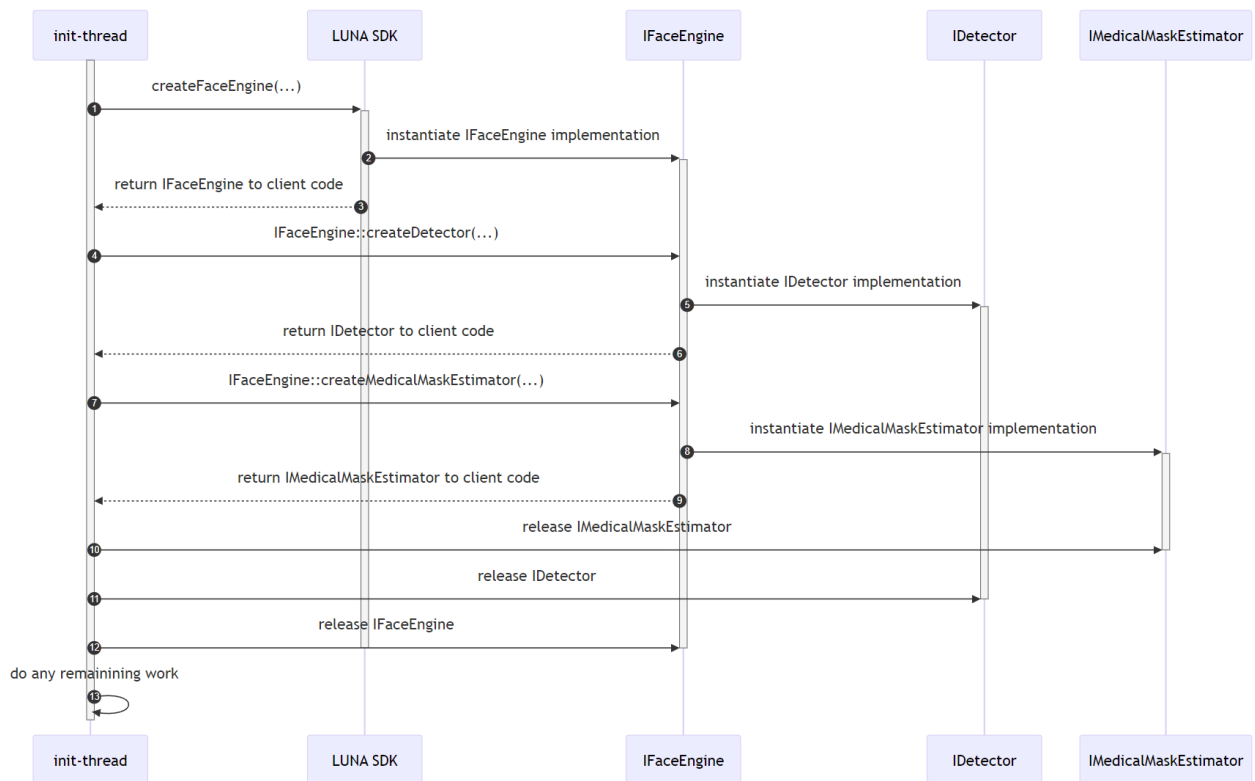
```
//warning: a correct, but not a good example due to these global variables
fsdk::IFaceEnginePtr faceEngine = fsdk::createFaceEngine("./data");
fsdk::IDetectorPtr detector = faceEngine->createDetector();
fsdk::IBestShotQualityEstimator bestShotQualityEstimator = faceEngine->
    createBestShotQualityEstimator();

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 `typedefs` for Ref's to all reference counted types are declared. All of them match the following naming convention: `InterfaceNamePtr`. 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 *SubImages* longer than it is necessary to complete data modification. See the documentation of the *map()* function family for details.

The supports IO routines 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 *Images* 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.

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 Data Paths

3.3.1 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.

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.

3.3.2 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.

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.

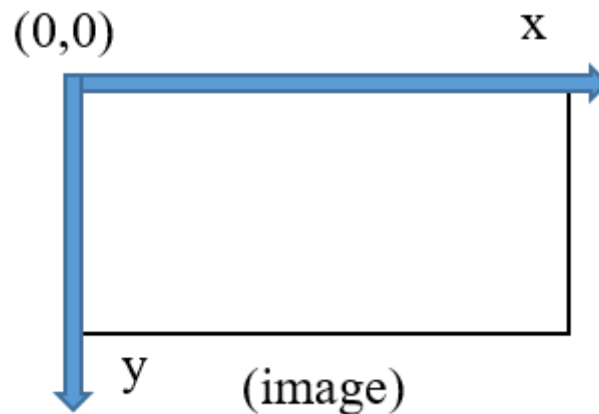


Figure 1: Source image coordinate system

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 its 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 Face Alignment

4.3.4.1 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.

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.

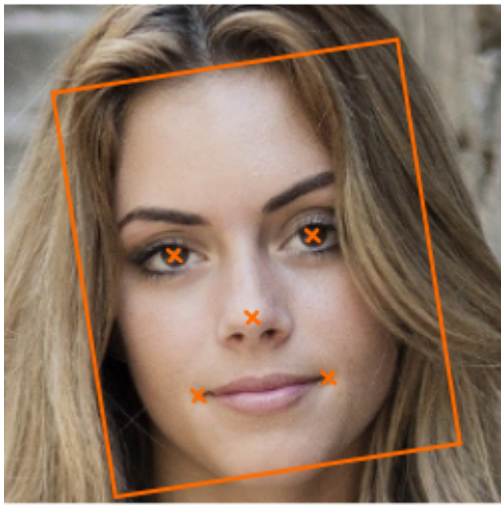


Figure 2: Face warping

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 Best shot selection functionality

6.2.1 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.

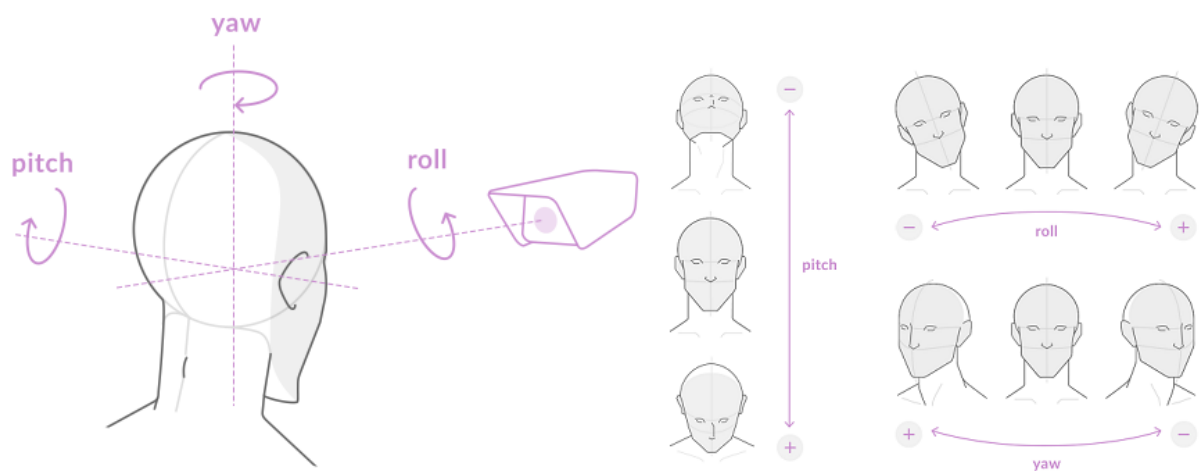


Figure 3: Head pose

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 1: “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

6.2.2 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 4: Blurred image (left), not blurred image (right)



Figure 5: Dark image (left), good quality image (right)



Figure 6: Light image (left), good quality image (right)



Figure 7: Image with uneven illumination (left), image with even illumination (right)



Figure 8: Image with specularities - 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-exposed (i.e., too dark);
- Image is over-exposed (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 blurred.
    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 blurred.
    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 2: “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.

```
<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.
```

6.2.3 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 9: Blurred image (left), not blurred image (right)



Figure 10: Dark image (left), good quality image (right)



Figure 11: Light image (left), good quality image (right)



Figure 12: Image with uneven illumination (left), image with even illumination (right)



Figure 13: Image with specularities - 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 blurred.
    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 SubjectiveQualityEstimator::Settings section. By default, these threshold values are set to optimal.

Table 3: “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

6.3 Face features extraction functionality

6.3.1 Eyes Estimation

Name: EyeEstimator

Algorithm description:

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

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 $x_{right} < x_{left}$.

`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

6.4 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 4)

Note: Prediction precision decreases as a rotation angle increases. We present typical average errors for different angle ranges in the table 4.

Table 4: “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°

	Range	-45°...+45°	< -45° or > +45°
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 14 for a reference.

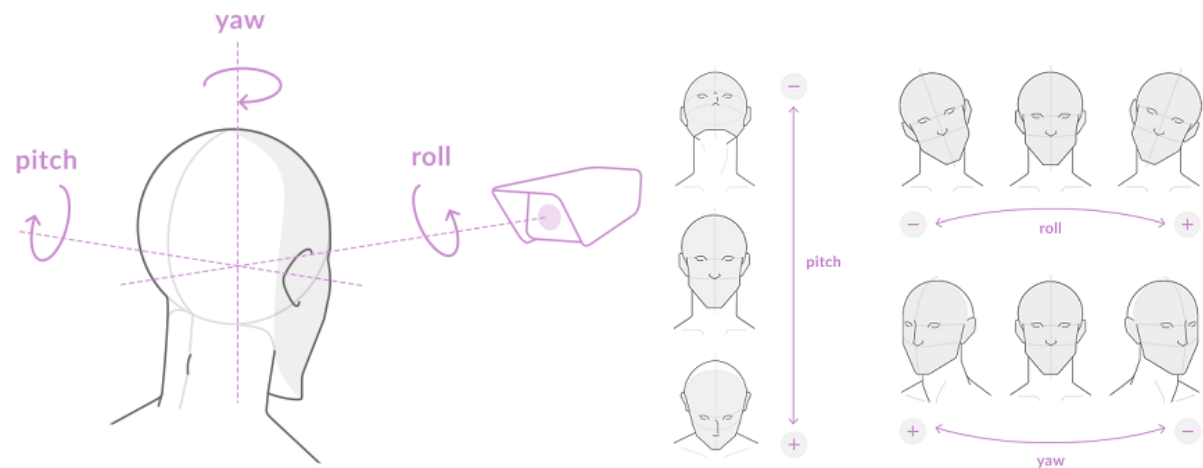


Figure 14: Head pose illustration

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.

6.5 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.

6.6 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 5: “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 6: “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

6.7 Liveness check functionality

6.8 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

Parameters	Requirements
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 8: “LivenessOneShotRGB estimator recommended thresholds”

Threshold	Recommended value
mobileRealThreshold	0.5
mobileQualityThreshold	0.5
mobileCalibrationCoeff	0.9967
mobileCalibrationCoeff_v12	0.9967

Configurations:

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

API structure name:

ILivenessOneShotRGBEstimator

Plan files, version 12:

- oneshot_rgb_liveness_v12_model_6_arm.plan

Plan files, version 13:

- oneshot_rgb_liveness_v13_model_6_arm.plan

```
## Mouth Estimation Functionality {#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 teeth exposed
    SmileOpen //!< smile with teeth 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 9: “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

6.9 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

6.10 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.

6.10.1 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.

6.10.2 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.

6.10.3 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:

1. 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
```

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 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.

6.10.4 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
};
```

6.10.5 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.

6.10.6 Filtration parameters

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

Table 14: “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

7 Descriptor processing facility

7.1 Overview

The section describes descriptors and all the processes and objects corresponding to them.

Descriptors and extraction facility is available only in the Complete edition only!

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.

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).

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”](#)).

7.2.2 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.

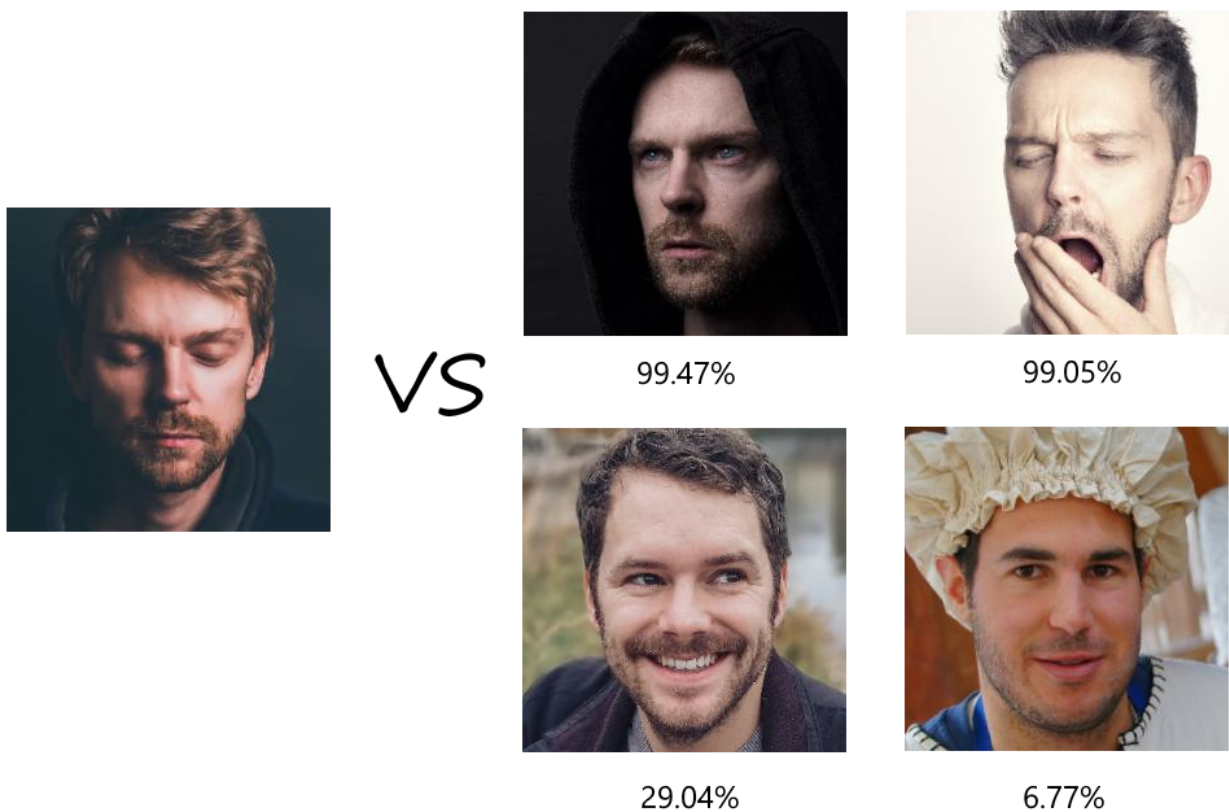


Figure 15: Matching

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.

8 System Requirements

8.1 IOS installations

FaceEngine requires:

- iOS version 12.0.

For development:

- XCode = 15.4.
- Compiler = AppleClang 15.0.0.15000309.

9 Hardware requirements

9.1 Mobile installations

Table 15: Models provided in distribution package and supported devices.

Neural network	ARM
FaceDet_v2_<detector_type> <i>first</i> <device>.plan	yes
FaceDet_v2_<detector_type> <i>second</i> <device>.plan	yes
FaceDet_v2_<detector_type> <i>third</i> <device>.plan	yes
headpose_v4_<device>.plan	yes
ags_v3_<device>.plan	yes
eyes_estimation_flwr8_<device>.plan	yes
eye_status_estimation_<device>.plan	yes
mask_clf_v3_<device>.plan	yes
mouth_estimation_v4_<device>.plan	yes
face_occlusion_v1_<device>.plan	yes
model_subjective_quality_v1_<device>.plan	yes
model_subjective_quality_v2_<device>.plan	yes
glasses_estimation_flwr_v2_<device>.plan	yes
onshot_rgb_liveness_v12_model_<model_id>_<device>.plan	yes
onshot_rgb_liveness_v13_model_<model_id>_<device>.plan	yes
vlTracker_detection_<device>.plan	yes

Neural network	ARM
vlTracker_template_<device>.plan	yes
vlTracker_update_<device>.plan	yes

oneshot_rgb_liveness_v12 is bundled in a separate archive

9.1.1 CPU requirements

arm64 is provided within iOS frameworks.

Bitcode-enabled libraries are available for iOS.

9.1.2 Memory requirements

RAM requirements are given for common for mobile platform verification pipeline.

Storage is amount of space specific version of installation takes on device. For iOS app thinning before deployment is assumed. As the result *.frameworks files in your final app archive will occupy (up to 30-60%, depending on platform) less storage space compared to ones found in the distribution.

Table 16: “Memory requirements”

Requirements for	iOS
RAM	400 MB
Storage Full	200 MB
Storage Frontend	170 MB

9.1.3 Number of threads on mobile devices

The description of according settings you can find in “Configuration Guide - Runtime settings”. The setting `<param name="numThreads" type="Value::Int1"x="-1"/>` means that will be taken the maximum number of available threads. 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.

10 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.

10.1 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.

10.2 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 Device-specific constraints

11.1 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.).

12 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

12.1 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>

12.2 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.

12.3 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.

12.4 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.

13 Useful tools

13.1 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.

13.1.1 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.

13.2 Metrics for performance analysis

13.2.1 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.

13.2.1.1 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).

13.3 Performance test parameters

Below are additional command line parameters that allow you to customize performance test operation.

13.3.1 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.

13.3.2 Batch and sensor parameters

Parameter	Description
<code>-b, --batch</code>	Sets the batch size (named parameter).
<code>-s, --sensor</code>	Sets the sensor type, for example, for the EyesBatch test.
<code>-y, --yuv</code>	Sets the YUV image for <i>YUV12toRGB</i> and <i>YUV21toRGB</i> tests.

13.3.3 iOS-specific parameters

Parameter	Description
<code>--data</code>	Path to the data directory (used only in non-standard iOS mode).
<code>--threads</code>	Number of threads used for testing (used only in non-standard iOS mode).
<code>--descriptor-model</code>	Specifies the descriptor model used in tests.
<code>--detector-type</code>	Specifies the detector type used in tests.

13.3.4 Stopping condition parameters

Parameter	Description
<code>--max-rel-height</code>	Threshold for relative height of the last step. If exceeded, stopping conditions are not met.
<code>--min-step-width</code>	Minimum width of the last step. If narrower, stopping conditions are not met.
<code>--max-rel-slope</code>	Threshold for relative slope of the last step. Combines the effects of <code>--max-rel-height</code> and <code>--min-step-width</code> .
<code>--min-steps</code>	Minimum number of steps required before stopping conditions can be evaluated.
<code>--min-iters</code>	Minimum number of iterations required before stopping.

Parameter	Description
<code>--max-iters</code>	Maximum number of iterations allowed (emergency stop condition).
<code>--max-time</code>	Maximum total execution time allowed (emergency stop condition).

13.3.5 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.

13.4 Stopping conditions

13.4.1 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.

13.4.2 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.

13.4.2.1 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.

13.4.3 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.

13.5 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 16: 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.

13.5.1 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.

13.5.2 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).

13.5.3 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.

13.5.4 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`.

13.5.5 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.

13.5.6 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`.

13.5.7 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.

13.6 Performance test challenges

13.6.1 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.

13.6.2 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.

13.6.3 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.

13.6.4 Test execution duration

The primary new challenge for performance tests is the significant amount of time required to gather a sufficient sample of data.

13.6.5 Artificial constraints efficiency

Artificial constraints via `--max-iters` or `--max-time` reduce the test's effectiveness by limiting the dataset size, potentially compromising reliability.

13.6.6 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.

13.7 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.

13.8 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.

14 Appendix A. Specifications

14.1 Runtime performance for mobile environment

Face detection performance depends on input image parameters, including resolution, bit depth, and the size of the detected face. The iOS platform uses Mobilenet by default.

Input data characteristics:

- Image resolution: 640x480px
- Image format: 24 BPP RGB

14.1.1 IOS

Performance measurements for the ARM architecture of the iPhone 11 are presented in the tables below. The measured values represent averages of at least 60 experiments. Mobilenet is used by default. The auto setting for the number of threads indicates that the maximum number of available threads will be used. To enable this mode, set the numThreads parameter to -1 in runtime.conf. This configuration corresponds to the number of available processor cores. We strongly recommend that you follow this guideline. Otherwise, performance may be significantly reduced. You can find descriptions of the corresponding settings in “Configuration Guide - Runtime settings”.

The tables below show performance metrics for the iPhone 11 (128 GB).

14.1.1.1 Matcher performance

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
59	1	1	6331780.0	361
59	auto	1	6285700.0	361
60	1	1	6297620.0	361
60	auto	1	6265170.0	361
65	1	1	6140410.0	324
65	auto	1	6204540.0	324

14.1.1.2 Extractor performance

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
59	1	1	46.55	144
59	auto	1	45.82	156

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
59	auto	4	48.0	180
59	auto	8	50.47	255
60	1	1	45.47	294
60	auto	1	45.88	317
60	auto	4	45.32	339
60	auto	8	45.47	361
65	1	1	37.11	255
65	auto	1	37.08	277
65	auto	4	36.74	299
65	auto	8	37.84	321

14.1.1.3 Detector performance

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
Detector	1	1	22.61	91
Detector	auto	1	22.58	96
Detector	auto	4	23.35	118
Detector	auto	8	23.27	144

14.1.1.4 Estimations performance with batch interface

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
AGS	1	1	0.81	38
AGS	auto	1	0.67	42
AGS	auto	4	0.51	48
AGS	auto	8	0.47	55
BestShotQuality	1	1	0.94	55
BestShotQuality	auto	1	0.94	59
BestShotQuality	auto	4	0.93	67

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
BestShotQuality	auto	8	0.92	75
Eyes	1	1	1.82	143
Eyes	auto	1	1.89	143
Eyes	auto	4	1.8	143
Eyes	auto	8	1.79	143
Glasses	1	1	2.47	363
Glasses	auto	1	2.47	363
Glasses	auto	4	2.42	363
Glasses	auto	8	2.41	363
HeadPose	1	1	0.54	363
HeadPose	auto	1	0.54	363
HeadPose	auto	4	0.53	363
HeadPose	auto	8	0.55	363
MedicalMask	1	1	14.78	363
MedicalMask	auto	1	14.97	363
MedicalMask	auto	4	14.22	363
MedicalMask	auto	8	14.07	363
Subjective Quality	1	1	2.77	363
Subjective Quality	auto	1	2.77	363
Subjective Quality	auto	4	2.74	363
Subjective Quality	auto	8	2.74	363
Warper	1	1	1.11	363
Warper	auto	1	1.06	363
Warper	auto	4	1.02	363
Warper	auto	8	1.19	363
LivenessOneShotRGBEstimat	1	1	171.03	363
LivenessOneShotRGBEstimator	auto	1	171.69	363
LivenessOneShotRGBEstimat	auto	4	171.97	363

Measurement	CPU threads	Batch Size	Percentile 95 (ms)	RAM Memory (Mb)
LivenessOneShotRGBEstimator	auto	8	177.86	565
Mouth	1	1	19.07	570
Mouth	auto	1	19.05	570
Mouth	auto	4	19.01	570
Mouth	auto	8	18.69	575
FaceOcclusion	1	1	16.39	575
FaceOcclusion	auto	1	17.44	576
FaceOcclusion	auto	4	16.1	577
FaceOcclusion	auto	8	16.23	579

14.2 Descriptor size

The table below shows size of serialized descriptors to estimate memory requirements.

Table 28: “Descriptor size”

Descriptor version	Data size (bytes)	Metadata size (bytes)	Total size
CNN 60	512	8	520
CNN 65	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.

14.3 Feature matrix

Mobile versions come only in the complete edition.

The table below shows FaceEngine features supported by the complete edition for mobile platforms.

Table 29: “Feature matrix”

Facility	Module	Complete
Core		Yes
Face detection & alignment	Face detector	Yes
Parameter estimation	BestShotQuality estimation	Yes
	Color estimation	Yes
	Eye estimation	Yes
	Head pose estimation	Yes
	AGS estimation	Yes
	LivenessOneShotRGB estimation	Yes
	Medical Mask estimation	Yes
	Quality estimation	Yes
	Mouth estimation	Yes
	Glasses estimation	Yes
Face descriptors	Descriptor extraction	Yes
	Descriptor matching	Yes
	Descriptor batching	Yes
	Descriptor search acceleration	Yes

See file “doc/FeatureMapMobile.htm” for more details.

15 Appendix B. Glossary

Table 30: 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

15.1 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.

15.2 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).

15.3 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.