



# **VisionLabs LivenessEngine Handbook**

## Contents

<b>Introduction</b>	<b>3</b>
<b>Glossary</b>	<b>4</b>
<b>1 LivenessEngine structure overview</b>	<b>5</b>
<b>2 Core facility</b>	<b>6</b>
2.1 LivenessEngine object . . . . .	6
2.1.1 Data paths . . . . .	6
2.1.2 Configuration file . . . . .	6
<b>3 Liveness facility</b>	<b>7</b>
3.1 Overview . . . . .	7
3.2 Liveness facility architecture . . . . .	7
3.3 Coordinate system . . . . .	7
3.4 Liveness types . . . . .	8
3.4.1 Simple liveness . . . . .	8
3.4.1.1 Basic liveness . . . . .	8
3.4.1.2 Angle liveness . . . . .	9
3.4.1.3 Mouth liveness . . . . .	10
3.4.1.4 Eyes liveness . . . . .	11
3.4.1.5 Eyebrows liveness . . . . .	11
3.4.1.6 Flow liveness . . . . .	11
3.4.1.7 Smile liveness . . . . .	12
3.4.1.9 Infrared liveness . . . . .	12
3.4.1.10 Unified liveness . . . . .	12
3.4.2 Complex liveness . . . . .	13
3.4.2.1 Depth liveness . . . . .	13
3.4.2.2 Error Codes . . . . .	13
<b>4 How to use</b>	<b>15</b>
<b>5 System requirements</b>	<b>17</b>
<b>6 Appendix A. Configuration file description</b>	<b>18</b>
<b>7 Appendix B. FAQ</b>	<b>22</b>
<b>8 Appendix C. LivenessEngine and models</b>	<b>23</b>
<b>9 Appendix D. Changelog</b>	<b>24</b>

## Introduction

LivenessEngine is a wrapper library with added functionality, which utilizes **FaceEngine** building blocks to produce different solutions for liveness detection problem. This short guide describes product's core concepts, represents main LivenessEngine features and suggests usage scenarios.

It is strongly recommended to familiarize with **FaceEngine Handbook** ( **FaceEngine\_Handbook.pdf** in the delivery package) before reading this guide since FaceEngine core concepts are widely used all over this handbook without additional explanation.

**Note!** This document is not a full-featured API reference manual nor a step by step tutorial. For reference pages, see **Doxygen API** documentation that is delivered with LivenessEngine (/doc/lSDK folder of the package).

## Glossary

Term	Definition
Liveness	Face features that allow distinguishing a living person from a photo/prearranged video.

## 1 LivenessEngine structure overview

LivenessEngine is subdivided into two facilities:

- **Core facility.** This facility is responsible for normal functioning of all other facilities. The core facility provides settings accessors and common interfaces. It also contains the main LivenessEngine root object that is used to create instances of all liveness test scenarios.
- **Liveness detection facility.** This facility is dedicated to liveness determination which is performed by different algorithms.

Both facilities are a set of classes dedicated to some common for them (yet specific) problem domain. Core facility is used for liveness detection facilities configuration and production.

## 2 Core facility

### 2.1 LivenessEngine object

The **LivenessEngine object** is a root object of the entire LivenessEngine. Everything begins with it, so it is essential to create at least one instance of it. Although it is possible to have multiple instances of the LivenessEngine, it is impractical to do so. Call **createLivenessEngine()** function to create a LivenessEngine instance. Also, you may specify default **dataPath** in **createLivenessEngine()** parameters.

The **LivenessEngine** requires **Face Engine** instance as well for internal sub-module initialization. If no **Face Engine** is provided, a new one is created implicitly.

#### 2.1.1 Data paths

Various LivenessEngine modules may require data files to operate. These files contain various constants used at runtime. The data directory location is assumed to reside in:

- /opt/visionlabs/data for the Linux CentOS delivery package;
- ./data for the Windows delivery package.

One may override the data directory location using **setDataDirectory()** method which is available in **ILivenessEngine**. Current data location may be retrieved via **getDataDirectory()** method

#### 2.1.2 Configuration file

For proper functioning LivenessEngine requires the configuration file. The file is called **livenessengine.conf** and stored in **dataPath** directory by default.

It contains configuration parameters which are required for every liveness scenario included in the library (face tracking configuration, each test scenario duration, various thresholds and so forth). More details on configuration file contents are in Appendix A.

At runtime, the configuration file data is managed by a special object that implements **ISettingsProvider** interface. The provider is instantiated using **createSettingsProvider()** function that accepts configuration file location as a parameter or uses a default one if not explicitly defined.

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

## 3 Liveness facility

### 3.1 Overview

Liveness detection facility is responsible for the determination whether or not a living person is in the image sequence. By image sequence here we mean a video stream from the camera or a local video file.

Liveness facility contains liveness detection algorithm structures implemented as state machines. These machines change their state each time **update()** function is called. Combined with **FaceEngine descriptor processing facility** it can be transformed into the powerful tool for user authentication.

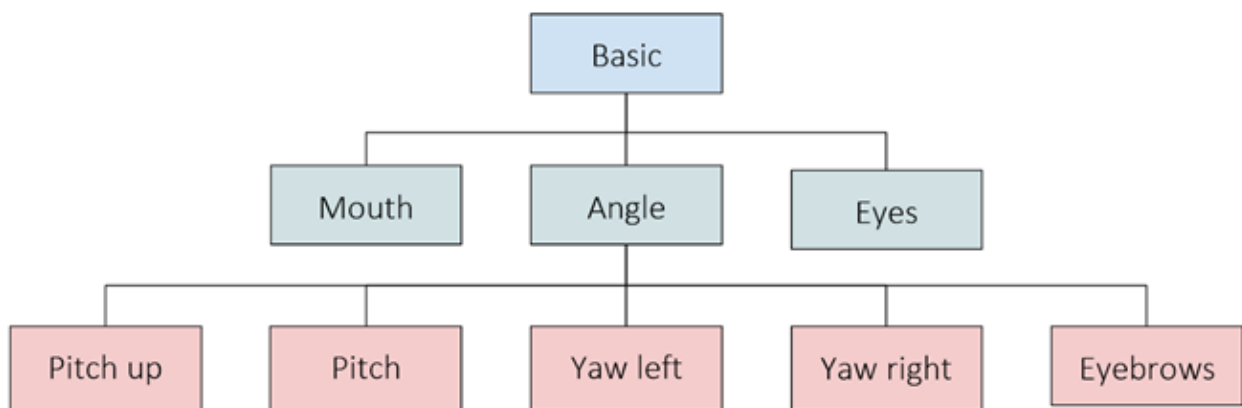
Liveness detection structure can be created via

**createLiveness()**, **createUnifiedLiveness()**, **createComplexLiveness()**

methods for simple, unified and complex liveness types respectively.

### 3.2 Liveness facility architecture

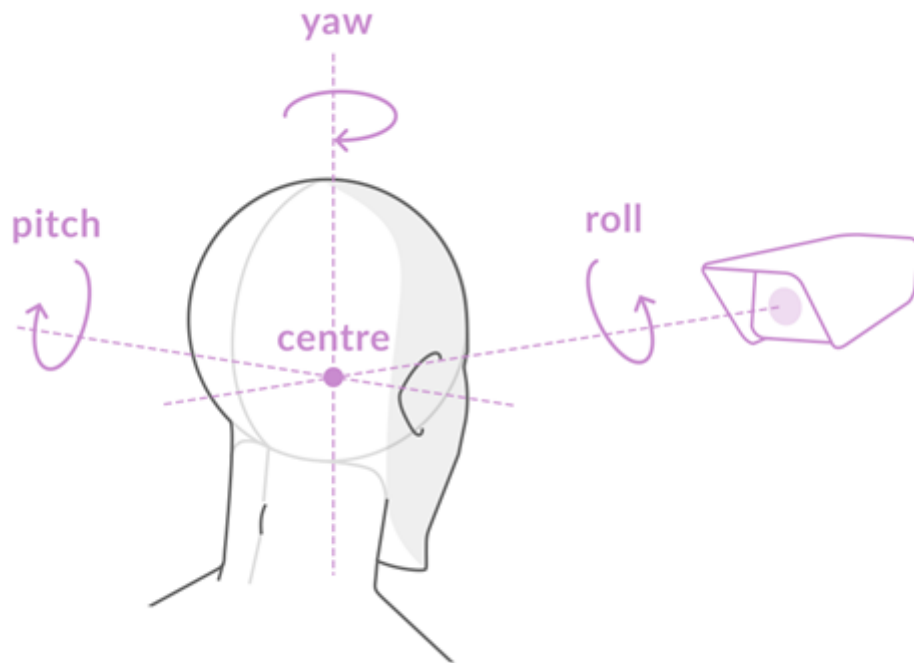
Liveness types are implemented according to the inheritance architecture.



**Figure 1:** Liveness architecture

### 3.3 Coordinate system

LivenessEngine uses the three-dimensional coordinate system. The coordinate system's center is represented in the image below.



**Figure 2:** Coordinate system midpoint

Actions required in liveness scenarios are performed regarding the accepted coordinate system (for example, “left turn” equals to counterclockwise head spin around the vertical axis). Graphical illustrations in the next chapters provide a visual representation of each scenario for better understanding.

### 3.4 Liveness types

Implemented liveness classes are divided into Simple and Complex types.

#### 3.4.1 Simple liveness

These liveness types require single video sequence for operation. Frames should be in **R8G8B8** format. Refer to **FaceEngine Handbook** for additional information about `fsdk::Image` structure. All simple liveness tests have common interface represented as `ILiveness` structure.

##### 3.4.1.1 Basic liveness

Each liveness type is inherited from the basic liveness class which utilizes a generic execution cycle and performs such common tasks as:



- basic initialization;
- face detection;
- additional data extraction / calculation;
- face tracking analysis;
  - using detection rectangles;
  - using landmark points;
- state change.

Each liveness test traces and analyzes primary for the test estimated attribute, and produces some result. The result is positive if a user succeeds in the correct alteration of the attribute. Otherwise, the result is negative.

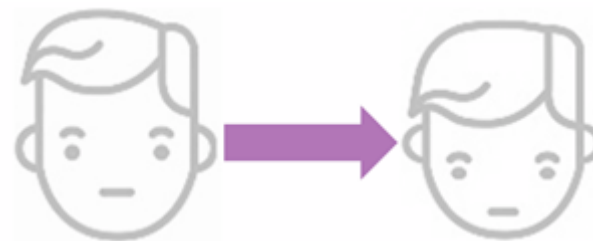
#### 3.4.1.2 Angle liveness

Angle liveness additionally performs head pose estimation. Refer to **FaceEngine Handbook** (chapter “Parameter estimation facility” section “Head pose estimation”) for more information on angle estimation.

Angle liveness types are the following:

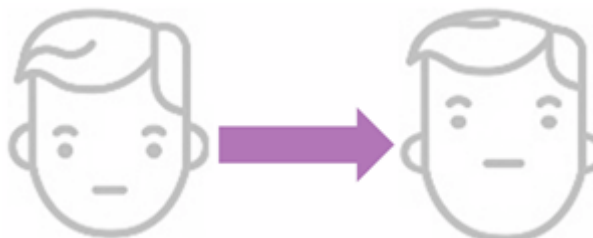
##### 1. Pitch angle

- a. **Nod scenario** means smooth head tilt in a positive direction until the required threshold is exceeded.



**Figure 3:** Head tilt in a positive direction

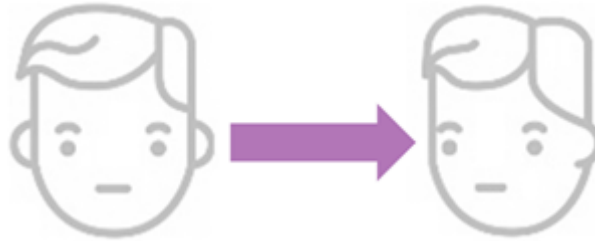
- b. **Head raise** scenario requires smooth head tilt in a negative direction until the required threshold is exceeded.



**Figure 4:** Head tilt in a negative direction

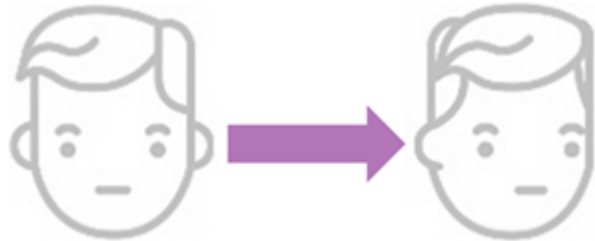
### 1. Yaw angle

- c. **Left turn** scenario requires smooth head rotation in a positive direction until the required threshold is exceeded.



**Figure 5:** Smooth head rotation in a positive direction

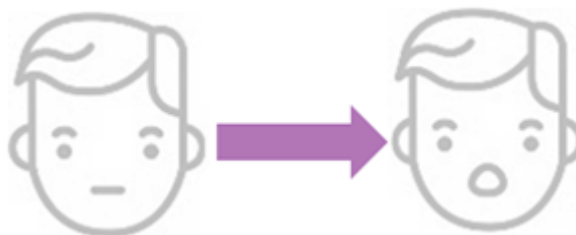
- d. **Right turn** scenario requires smooth head rotation in a negative direction until the required threshold is exceeded.



**Figure 6:** Smooth head rotation in a negative direction

#### 3.4.1.3 Mouth liveness

Mouth liveness performs mouth landmarks analysis. In this scenario distance between mouth landmarks increases until the required threshold is exceeded (i.e. a user opens a mouth).



**Figure 7:** Mouth liveness

Refer to **FaceEngine Handbook** for more information on face alignment.

#### 3.4.1.4 Eyes liveness

Eyes liveness performs eye state estimation and analysis. In this scenario a user should blink, i.e., both eyes are opened, closed and opened again simultaneously.

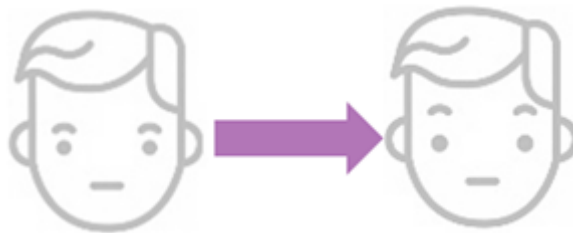


**Figure 8:** Eyes liveness

Refer to **FaceEngine Handbook** (Chapter “Parameters estimation facility” section “Eyes estimation”) for more information on eyes estimation.

#### 3.4.1.5 Eyebrows liveness

Eyebrow liveness performs eyebrow landmarks analysis. This scenario requires increase of the distance between eyebrows and eyes landmarks (i.e. eyebrow rising) until the required threshold is exceeded.

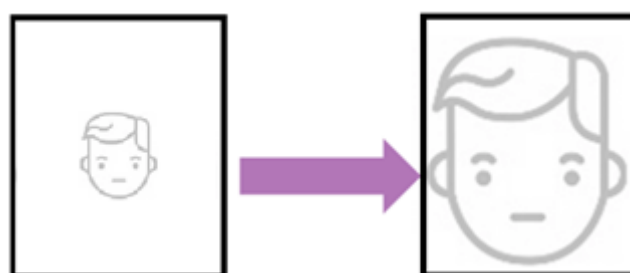


**Figure 9:** Eyebrow liveness

Refer to **FaceEngine Handbook** (chapter “Face detection facility” section “Face alignment”) for more information on face alignment.

#### 3.4.1.6 Flow liveness

Flow liveness performs optical flow analysis. This scenario requires a smooth increase of face detection rectangle area to obtain a required number of frames to calculate the optical flow.



**Figure 10:** Flow liveness

This liveness type is designed for mobile phones, and the results may be erroneous on other platforms.

#### 3.4.1.7 Smile liveness

Smile liveness performs face warp analysis. This scenario requires user to smile until the probability calculated by neural network will be above threshold, specialized by configuration file.

#### 3.4.1.8

#### 3.4.1.9 Infrared liveness

Infrared liveness performs face warp analysis using image acquired from infrared camera. This scenario requires user to normally appear in front of a camera until the probability calculated by neural network will be above threshold, specialized by configuration file.

**Note:** Most Infrared cameras provide 1-channel grayscale image, such image should be converted to 3-channel grayscale image before passing into infrared liveness.

#### 3.4.1.10 Unified liveness

Unified liveness combines previous types algorithms with the exception of flow and blink types, and apart from tracking of main entity, which was specialized on creation stage, performs additional calculation and analysis in order to detect fraud attempts.

Calculated and tracked entities:

- angles;
- mouth landmarks distance;
- eyebrow landmarks distance;
- eye states (blinks);
- smile probability;

Rigidity of fraud tracking is set by configuration parameters. User can adjust fraud checking by enabling or disabling additional verifications with **consider** method.

The main action scenario is borrowed from simple liveness type at creation stage, however user should not perform any actions except main one, because it will be considered as a fraud attempt.

### 3.4.2 Complex liveness

These liveness types require additional unordinary data for analysis. Such data cannot be obtained by common rgb camera, so it requires the use of complementary devices for operating.

#### 3.4.2.1 Depth liveness

Currently depth liveness is the only complex type that is supported. It requires 16 bit depth matrix, which contains information relating to the distance of the surfaces of scene objects from a viewpoint in millimeters. For correct operation face should be placed on distance from 0.5 to 4.5 meters.

This liveness type does not require any actions because it performs depth map face region of interest analysis using neural networks. For additional information refer to Doxygen API documentation that is delivered with LivenessEngine.

Rgb image and 16 bit depth map must be aligned.

#### 3.4.2.2 Error Codes

The LSDKError structure (possible return codes).

LSDKError code	Description
Ok	Ok.
NotInitialized	Liveness not initialized.
NotReady	Liveness not ready, require more updates.
PreconditionFailed	Starting condition is not accomplished, liveness not started yet.
Internal	Internal error.

The most common error state is `PreconditionFailed`. And related to the liveness type, interpretation may be different. Possible options are below (details in `include/lsdk/ILiveness.h`).

Liveness type	PreconditionFailed interpretation
LA_PITCH_DOWN	Yaw, pitch or roll absolute value larger than expected value.
LA_PITCH_UP	Yaw, pitch or roll absolute value larger than expected value.
LA_YAW_LEFT	Yaw, pitch or roll absolute value larger than expected value.

Liveness type	PreconditionFailed interpretation
LA_YAW_RIGHT	Yaw, pitch or roll absolute value larger than expected value.
LA_SMILE	Smiling score is less than 0.0.
LA_MOUTH	The currentDistance value less than expected value.
LA_EYEBROW	Yaw, pitch or roll absolute value larger than expected value.
LA_EYE	State of eyes aren't open!
LA_FLOW	Detected face rect's sides ratio is greater than expected value.
LA_INFRARED	IR estimation score is less than 0.0.

And at least the result value PreconditionFailed may be related with detection rectangle. Before starting liveness estimation, there is one more verification of detection rectangle, which is checkBorder. The checkBorder method checks if detection rectangle is inside allowed area. Otherwise, You can see logging message as below:

```
[30.08.2021 13:40:42] [Debug] [BasicLiveness] checkBorder failed! Detection
rectangle is not inside allowed area: x,y{4, 4} w,h{479, 463}
```

**Note** Find details about values for each Liveness type, mentioned above in data/livenessengine.conf. Please pay attention, level of log verbosity is 4 - Debug.

## 4 How to use

Each liveness test is implemented as state machine which modifies its state with each call of **update()** method. Current state can be identified by **error** value returned after update call. So in common scenario user should submit images for processing and analyze returned error.

Error types:

**ERR\_OK** - denotes that result is ready for acquisition. **ERR\_NOT\_INITIALIZED** - denotes that liveness test is not initialized, it happens in critical cases when **LivenessEngine** or **FaceEngine** core objects were not initialized. Make sure that data and configuration paths are correct and try to recreate core modules, or call **loadSettings()** method. **ERR\_NOT\_READY** - denotes that result is not ready and additional update calls are required. **ERR\_PRECONDITION\_FAILED** - denotes that test has not yet begun and preconditions are not done (Preconditions can vary depending on liveness type). **ERR\_INTERNAL** - denotes that internal error occurred, please dump input data and contact your SDK advisor.

Right error sequence described below:

ERR\_NOT\_INITIALIZED -> ERR\_PRECONDITION\_FAILED -> ERR\_NOT\_READY -> ERR\_NOT\_READY -> ... -> ERR\_NOT\_READY -> ERR\_OK

The following is the common usage of the liveness detector types:

1) Initialize video capture.

2) Create FaceEngine and LivenessEngine structures:

```
auto resFaceEngine = createFaceEngine(...);  
if (!resFaceEngine) { ... }  
IFaceEnginePtr faceEngine = resFaceEngine.getValue();  
auto resLivenessEngine =  
createLivenessEngine(faceEngine, ...);  
if (!resLivenessEngine) { ... };  
auto livenessEngine = resLivenessEngine.getValue();
```

3) Create required liveness test type:

```
auto resLiveness = livenessEngine->createLiveness(LivenessAlgorithmType::LA_INFRARED);  
if (!resLiveness) { ... };  
auto liveness = resLiveness.getValue();
```

4) In cycle acquire image from video stream, preprocess it if necessary.

5) Submit photos and check result status:

```
ResultValue<LSDKError, bool> result = liveness->update(img);
```

In case of complex liveness:

```
ResultValue<LSDKError,bool>result = depthLiveness->update(color,depth);
```

6) If result is not erroneous retrieve result:

```
if (result.getResult() == LSDKError::ERR_OK)
```

```
liveness = result.getValue();
```

For more detailed information about liveness usage please refer to **example\_liveness** and **example\_depth** located at /examples folder of the LivenessEngine package.



## 5 System requirements

LivenessEngine requires FaceEngine version 3.0.1+ for proper operation.

Other system requirements (supported OS, compilers and so forth) are similar to FaceEngine (see chapter 9 for additional information).

## 6 Appendix A. Configuration file description

The configuration file has a module-based structure. Each liveness algorithm has:

- basic parameters which are extracted from **Liveness::Basic** section;
- a set of unique parameters which are extracted from corresponding Liveness::<LivenessType> section.

**Table 4:** Common parameters.

Parameter name	Description	Default value
maxRuntime	Parameter is responsible for the liveness scenario's length (in frames). If liveness is not verified during amount of frames less or equal to <b>maxRuntime</b> value, liveness scenario is failed.	140
successLength	Parameter is responsible for the required length of successful actions in the liveness test (in frames). For example, angle liveness test is passed if amount of frames with angle higher than <b>angleThreshold</b> exceeds the <b>successLength</b> .	5

**Table 5:** Parameters for tracking analysis.

Parameter name	Description	Default value
trackAreaThreshold	Sets face detection area alteration threshold. If the difference between detection areas on the previous and the current frame is higher than the frame area multiplied by threshold, track loss is detected.	1
trackCenterThreshold	Sets face detection rectangle center alteration threshold. If the difference between rectangle centers on the previous and the current frame is higher than the frame width multiplied by threshold, track loss is detected.	1
allowedTrackLoss	Sets the number of frames in a row that can have no track on them. If amount of frames exceeds the threshold, a liveness scenario is failed.	20

Parameter name	Description	Default value
verticalPadding	Sets vertical padding for allowed area on image. As starting condition face should be placed in formed frame. Result frame height = input image height - $2 \times \text{verticalPadding}$ input image height.	0.01
horizontalPadding	Sets horizontal padding for allowed area on image. As starting condition face should be placed in formed frame. Result frame width = input image width - $2 \times \text{horizontalPadding}$ input image width.	0.01
allowMultipleFaces	Allows or forbids multiple faces on image, liveness test will fail, when more than one face will be detected, if this value is set to 0.	1
landmarkThreshold	Sets face landmarks alteration threshold. If the difference between mean landmarks alteration on the previous and the current frame is higher than the threshold, track loss is detected. If threshold is set to 0.0 landmark tracking is disabled.	0.0
redetectTolerance	Sets the tolerance which is used in redetection. Refer to <b>FaceEngine Handbook</b> (chapter “Face detection facility” section “Face detection”) for more information.	20

**Table 6:** Angle liveness parameters.

Parameter name	Description	Default value
angleThreshold	Sets the threshold of the corresponding angle which should be surpassed.	depends on the liveness type
angleRange	Sets the initial range of all angles required to start the test. For example, if roll is +5, pitch is +8 and yaw is -7, and <b>angleRange</b> is 10 the test can be started.	10.0

Parameter name	Description	Default value
heldAngleRange	Sets the range of not tracked angles, which should be maintained during the test. For example, if test requires pitch angle to be above 20, increasing/decreasing yaw above/less $\pm$ heldAngleRange will result test failure . <b>Used in unified liveness.</b>	15.0

**Table 7:** Eyebrow, Mouth, and Eye liveness parameters.

Parameter name	Description	Default value
successThreshold	Sets the final threshold of the measured value in corresponding tests. Similar to angleThreshold in angle liveness.	depends on the liveness type
startThreshold	Sets threshold which is used as starting condition for tracked attribute. In case of eyebrow liveness startThreshold is initialized with value calculated on first frame.	depends on the liveness type
fraudThreshold	Sets the threshold which is used to track fraud attempts. For example, if test requires pitch angle increase, and calculated mouth distance > mouth fraudThreshold then fraud attempt is detected and test results failure. <b>Used in unified liveness.</b>	depends on the liveness type

**Table 8:** Flow parameters.

Parameter name	Description	Default value
smallFaceThreshold	Image sequence starts with the face detection rectangle width lower than <b>(smallFaceThreshold * frame width)</b> . Required for flow liveness as a starting condition.	0.45
bigFaceThreshold	Image sequence finishes with the face detection rectangle width higher than <b>(bigFaceThreshold * frame width)</b> . Required for flow liveness as a finish condition.	0.7

Parameter name	Description	Default value
sequenceThreshold	Sets required number of images in sequence. Sequence length increase results increase of accuracy and processing time.	10

## 7 Appendix B. FAQ

1. **Q:** This document contains high-level descriptions and no code examples nor reference. Where can one find them?

**A:** The complete type and function reference are provided as an interactive web-based documentation; see the ***doc/lsdk/index.html*** inside the **LUNA SDK** package. The examples are located in the ***/examples*** folder of the package.

## 8 Appendix C. LivenessEngine and models

You can find all required information in the “LivenessEngine\_and\_models.htm” table.

## 9 Appendix D. Changelog

Date	Version	Notes
31.10.17	1	Initial release.
14.11.17	2	Added FAQ section.
13.12.17	3	Added complex liveness information.
20.12.17	4	Added unified liveness information, fixed configuration parameters description.
19.02.18	4.1	Additional information about types, added usage scenario, more links to doxygen and examples.
02.03.18	Appendix C	LivenessEngine and models