JanICE Documentation

Release 6.0.0

Noblis

Jun 27, 2018

CONTENTS

CHAPTER

ONE

CONCEPTS

1.1 Error Handling

The API handles errors using return codes. Valid return codes are defined *JaniceError*. In general, it is assumed that new memory is only allocated if a function returns JANICE_SUCCESS. Therefore, **implementors are REQUIRED to deallocate any memory allocated during a function call if that function returns an error.**

1.2 Memory Allocation

The API often passes unallocated pointers to functions for the implementor to allocate appropriately. This is indicated if the type of a function input is JaniceObject**, or in the case of a utility typedef JaniceTypedef*. It is considered a best practice for unallocated pointers to be initialized to NULL before they are passed to a function, but this is not guaranteed. It is the responsibility of the users of the API to ensure that pointers do not point to valid data before they are passed to functions in which they are modified, as this would cause memory leaks.

1.3 Thread Safety

All functions are marked one of:

1.3.1 Thread Safe

Can be called simultaneously from multiple threads, even when the invocations use shared data.

1.3.2 Reentrant

Can be called simultaneously from multiple threads, but only if each invocation uses its own data.

1.3.3 Thread Unsafe

Can not be called simultaneously from multiple threads.

1.4 Compiling

Define JANICE_LIBRARY during compilation to export JanICE symbols.

1.5 Versioning

This API follows the semantic versioning paradigm. Each released iteration is tagged with a major.minor.patch version. A change in the major version indicates a breaking change. A change in the minor version indicates a backwards-compatible change. A change in the patch version indicates a backwards-compatible bug fix.

CHAPTER

TWO

ERRORS

2.1 Overview

Every function in the JanICE *C* API returns an error code when executed. In the case of successful application JANICE_SUCCESS is returned, otherwise a code indicating the specific issue is returned. The error codes are enumerated using the *JaniceError* type.

2.2 Enumerations

2.2.1 JaniceError

The error codes defined in the JanICE *C* API

Code	Description
JANICE_SUCCESS	No error
JANICE_UNKNOWN_ERROR	Catch all error code
JANICE_INTERNAL_ERROR	Internal SDK error
JANICE_OUT_OF_MEMORY	Out of memory error
JANICE_INVALID_SDK_PATH	Invalid SDK location
JANICE_BAD_SDK_CONFIG	Invalid SDK configuration
JANICE_BAD_LICENSE	Incorrect license file
JANICE_MISSING_DATA	Missing SDK data
JANICE_INVALID_GPU	The GPU is not functioning
JANICE_BAD_ARGUMENT	An argument to a JanICE function is invalid
JANICE_OPEN_ERROR	Failed to open a file
JANICE_READ_ERROR	Failed to read from a file
JANICE_WRITE_ERROR	Failed to write to a file
JANICE_PARSE_ERROR	Failed to parse a file
JANICE_INVALID_MEDIA	Failed to decode a media file
JANICE_OUT_OF_BOUNDS_ACCESS	Out of bounds access into a buffer.
JANICE_MEDIA_AT_END	A media iterator has reached the end of its data.
JANICE_INVALID_ATTRIBUTE_KEY	An invalid attribute key was provided.
JANICE_MISSING_ATTRIBUTE	A value for a valid attribute key is not available.
JANICE_DUPLICATE_ID	Template id already exists in a gallery
JANICE_MISSING_ID	Template id can’t be found
JANICE_MISSING_FILE_NAME	An expected file name is not given
JANICE_INCORRECT_ROLE	Incorrect template role
JANICE_FAILURE_TO_SERIALIZE	Could not serialize a data structure
JANICE_FAILURE_TO_DESERIALIZE	Could not deserialize a data structure
JANICE_BATCH_ABORTED_EARLY	Batch call aborted early due to encountered error
JANICE_BATCH_FINISHED_WITH_ERRORS	Batch call finished but with errors.
JANICE_NOT_IMPLEMENTED	Optional function return
JANICE_NUM_ERRORS	Utility to iterate over all errors

2.3 Structs

2.3.1 JaniceErrors

A structure to represent a list of *JaniceError* objects.

Fields

Name	Type	Description
errors	*JaniceError**	An array of error objects.
length	size_t	The number of elements in `errors`

2.4 Functions

2.4.1 janice_error_to_string

Convert a *JaniceError* into a string for printing.

Signature

```
JANICE_EXPORT const char* janice_error_to_string(JaniceError error);
```

Thread Safety

This function is *Thread Safe*.

Parameters

Name	Type	Description
error	*JaniceError*	An error code

Return Value

This is the only function in the API that does not return *JaniceError*. It returns const char* which is a null-terminated list of characters that describe the input error.

2.4.2 janice_clear_errors

Free any memory associated with a *JaniceErrors* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_errors(JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
errors	*JaniceErrors**	An errors objects to clear.

CHAPTER

THREE

INITIALIZATION

3.1 Enumerations

3.1.1 JaniceLogLevel

A enumeration to control the logging fidelity of JanICE applications. Possible log levels are:

Level	Description
JaniceLogDebug	Output fine-grained informational events useful for debugging.
JaniceLogInfo	Output course-grained events indicating progress.
JaniceLogWarning	Output warning events that might lead to a failure.
JaniceLogError	Output failure events that don't stop the application from running.
JaniceLogCritical	Output events that will cause the application to abort.

3.2 Structs

3.2.1 JaniceConfigurationItem

A key-value pair representing a single configuration setting

Fields

Name	Type	Description
key	char*	A null-terminated configuration key
value	char*	A null-terminated configuration value

3.2.2 JaniceConfiguration

A structure representing a list of *JaniceConfigurationItem* objects.

Fields

Name	Type	Description
values	*JaniceConfigurationItem**	An array of configuration objects
length	size_t	The number of elements in `values`

3.3 Functions

3.3.1 janice_initialize

Initialize global or shared state for the implementation. This function should be called once at the start of the application, before making any other calls to the API.

Signature

```
JANICE_EXPORT JaniceError janice_initialize(const char* sdk_path,
                                            const char* temp_path,
                                            const char* algorithm,
                                            const int num_threads,
                                            const int* gpus,
                                            const int num_gpus);
```

Thread Safety

This function is *Thread Unsafe*.

Parameters

Name	Type	Description
sdk_path	const char*	Path to a **read-only** directory containing the JanICE compliant SDK as specified by the implementor.
temp_path	const char*	Path to an existing empty **read-write** directory for use as temporary file storage by the implementation. This path must be guaranteed until *janice_finalize*.
algorithm	const char*	An empty string indicating the default algorithm, or an implementation defined containing an alternative configuration.
num_threads	const int	The number of threads the implementation is allowed to use. A value of '-1' indicates that the implementation should use all available hardware.
gpus	const int*	A list of indices of GPUs available to the implementation. The length of the list is given by `num_gpus`. If the implementor does not require a GPU in their solution they can ignore this parameter. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
num_gpus	const int	The length of the `gpus` array. If no GPUs are available this should be set to 0.

3.3.2 janice_set_log_level

Set the global log level for the implementation. By default, the log level is set to `JaniceLogWarning`.

Signature

```
JANICE_EXPORT JaniceError janice_set_log_level(JaniceLogLevel level);
```

Thread Safety

This function is *Thread Unsafe*.

Parameters

Name	Type	Description
level	*JaniceLogLevel*	The new log level for the implementation

3.3.3 janice_api_version

Query the implementation for the version of the JanICE API it was designed to implement. See *Versioning* for more information on the versioning convention for this API.

Signature

```
JANICE_EXPORT JaniceError janice_api_version(uint32_t* major,
                                             uint32_t* minor,
                                             uint32_t* patch);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
major	uint32_t*	The supported major version of the API. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
minor	uint32_t*	The supported minor version of the API. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
patch	uint32_t*	The supported patch version of the API. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

3.3.4 janice_sdk_version

Query the implementation for its SDK version.

Signature

```
JANICE_EXPORT JaniceError janice_sdk_version(uint32_t* major,
                                             uint32_t* minor,
                                             uint32_t* patch);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
major	uint32_t*	The major version of the SDK. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
minor	uint32_t*	The minor version of the SDK. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
patch	uint32_t*	The patch version of the SDK. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

3.3.5 janice_get_current_configuration

Get the current implementation configuration as a list of key value pairs

Signature

```
JANICE_EXPORT JaniceError janice_get_current_configuration(JaniceConfiguration*␣
↪configuration);
```

Parameters

Name	Type	Description
configuration	*JaniceConfiguration**	A list to hold the current configuration settings of the implementation. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_configuration*.

3.3.6 janice_finalize

Destroy any resources created by *janice_initialize* and finalize the application. This should be called once after all other API calls.

Signature

```
JANICE_EXPORT JaniceError janice_finalize();
```

Thread Safety

This function is *Thread Unsafe*.

3.3.7 janice_clear_configuration

Free any memory associated with a *JaniceConfiguration* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_configuration(JaniceConfiguration*␣
↪configuration);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
configuration	*JaniceConfiguration**	A configuration object to clear.

CHAPTER

FOUR

I/O

4.1 Overview

As a computer vision API it is a requirement that images and videos are loaded into a common structure that can be processed by the rest of the API. In this case, we strive to isolate the I/O functions from the rest of the API. This serves three purposes:

1. It allows implementations to be agnostic to the method and type of image storage, compression techniques and other factors
2. It keeps implementations from having to worry about licenses, patents and other factors that can arise from distributing proprietary image formats
3. It allows implementations to be "future-proof" with regards to future developments of image or video formats

To accomplish this goal the API defines a simple interface of two structures, *JaniceImage* and *JaniceMediaIterator* which correspond to a single image or frame and an entire media respectively. These interfaces allow pixel-level access for implementations and can be changed independently to work with new formats.

4.2 Structs

4.2.1 JaniceImage

A structure representing a single frame or an image

Fields

Name	Type	Description
channels	uint32_t	The number of channels in the image.
rows	uint32_t	The number of rows in the image.
cols	uint32_t	The number of columns in the image.
data	uint8_t*	A contiguous, row-major array containing pixel data.
owner	bool	True if the image owns its data and should delete it, false otherwise.

4.2.2 JaniceMediaIteratorState

A void pointer to a user-defined structure that contains state required for a *JaniceMediaIterator*.

4.2.3 JaniceMediaIterator

An interface representing a single image, a sparse selection of video frames or a full video. JaniceMediaIterator implements an iterator interface on media to enable lazy loading via function pointers.

is_video

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, bool* video)
```

The function sets `video` to True if `it` is a video. Otherwise, it sets `video` to False. `it` should be considered a video if multiple still images can be retrieved with successive calls to *next*. This function should return `JANICE_SUCCESS` if `video` can be set to True or False.

get_frame_rate

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, float* frame_rate)
```

The function sets `frame_rate` to the frame rate of `it`, if that information is available. If `frame_rate` can be set to a value this function should return `JANICE_SUCCESS`, otherwise it should return `JANICE_INVALID_MEDIA`. In the case of downsampling, this should return the observed frame rate.

get_physical_frame_rate

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, float* physical_frame_rate)
```

The physical frame rate is the actual frame rate of the video, independent of processing done by media iterator. The function sets `physical_frame_rate` to the physical frame rate of `it`, if that information is available. If `frame_rate` can be set to a value this function should return `JANICE_SUCCESS`, otherwise it should return `JANICE_INVALID_MEDIA`.

next

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, JaniceImage* img)
```

The functions sets `img` to the next still image or frame from `it` and and advances `it` one position. If `img` is successfully set this function should return `JANICE_SUCCESS`. If `it` has already iterated through all available images, this function should return `JANICE_MEDIA_AT_END`. Otherwise, a relevant error code should be returned.

seek

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, uint32_t frame)
```

The function sets the internal state of `it` such that a successive call to *next* will return the image with index `frame`. If `it` is an image, this function should work for `frame` == 0, in which case it is equivalent to *reset*, otherwise `JANICE_INVALID_MEDIA` should be returned. If `it` is a video, the implementation may optionally do bounds checking on `frame`. If the seek is successful, `JANICE_SUCCESS` should be returned.

get

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, JaniceImage* img, uint32_t frame)
```

This function gets a specific frame from `it` and stores it in `img`. It should not modify the internal state of `it`. If `it` is an image, this function should work or :code‘frame‘ == 0. If `frame` != 0 and `it` is an image, this function should return `JANICE_INVALID_MEDIA`. If `it` is a video, the implementation may optionally do bounds checking on `frame`. If the get is successful, this function should return `JANICE_SUCCESS`. If the get is not successful, an appropriate error code should be returned and `it` may be left in an undefined state.

tell

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, uint32_t* frame)
```

Get the current position of `it` and store it in `frame`. If `it` is an image, this function should return `JANICE_INVALID_MEDIA`. If `it` is a video and its position can be successfully queried, this function should return `JANICE_SUCCESS`. Otherwise, an appropriate error code should be returned.

reset

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it)
```

Reset `it` to an initial valid state. This function should return `JANICE_SUCCESS` if `it` can be reset, otherwise an appropriate error code should be returned.

physical_frame

A function pointer with signature:

```
JaniceError(JaniceMediaIterator* it, uint32_t frame, uint32_t* physical_frame)
```

Map an observed frame to a physical frame. If a mapping is possible this function should return `JANICE_SUCCESS`. Otherwise, an appropriate error code should be returned.

free_image

A function pointer with signature:

```
JaniceError(JaniceImage* img)
```

Free any memory associated with `img`. *free_image* should be called with the same iterator that allocated `img` with a call to either *next* or *get*. This function should return `JANICE_SUCCESS` if `img` is successfully freed, otherwise an appropriate error code should be returned.

free

A function pointer with signature:

```
JaniceError(JaniceMediaIterator** it)
```

Free any memory associated with `it`. This function should return `JANICE_SUCCESS` if `it` is freed successfully, otherwise and appropriate error code should be returned.

Fields

Name	Type	Description
is_video	*JaniceError*(*JaniceMediaIterator**, bool*)	See *is_video*.
get_frame_rate	*JaniceError*(*JaniceMediaIterator**, float*)	See *get_frame_rate*.
get_physical_frame_rate	*JaniceError*(*JaniceMediaIterator**, float*)	See *get_physical_frame_rate*.
next	*JaniceError*(*JaniceMediaIterator**, *JaniceImage**)	See *next*.
seek	*JaniceError*(*JaniceMediaIterator**, uint32_t)	See *seek*.
get	*JaniceError*(*JaniceMediaIterator**, *JaniceImage**, uint32_t)	See *get*.
tell	*JaniceError*(*JaniceMediaIterator**, uint32_t*)	See *tell*.
reset	*JaniceError*(*JaniceMediaIterator**)	See *reset*.
physical_frame	*JaniceError*(*JaniceMediaIterator**, uint32_t, uint32_t*)	See *physical_frame*.
free_image	*JaniceError*(*JaniceImage**)	See *free_image*.
free	*JaniceError*(*JaniceMediaIterator***)	See *free*.
_internal	*JaniceMediaIteratorState*	A pointer to memory meant for internal use only. The implementation may use this to store persistent state about the iterator.

4.2.4 JaniceMediaIterators

A structure representing a list of *JaniceMediaIterator* objects.

Fields

Name	Type	Description
media	*JaniceMediaIterator**	An array of media iterator objects.
length	size_t	The number of elements in `media`

4.2.5 JaniceMediaIteratorsGroup

A structure to represent a list of *JaniceMediaIterators* objects.

Fields

Name	Type	Description
group	*JaniceMediaIterators*	An array of media objects.
length	size_t	The number of elements in `group`

CHAPTER

FIVE

CONTEXT

A context is a single object for managing the various hyperparameters parameters required by JanICE functions.

5.1 Enumerations

5.1.1 JaniceDetectionPolicy

A policy that controls the types of objects that should be detected by a call to *janice_detect*. Supported policies are:

Policy	Description
JaniceDetectAll	Detect all objects present in the media.
JaniceDetectLargest	Detect the largest object present in the media. Running detection with this policy should produce at most one detection.
JaniceDetectBest	Detect the best object present in the media. The implementor is responsible for defining what “best” entails in the context of their algorithm. Running detection with this policy should produce at most one detection.

5.1.2 JaniceEnrollmentType

Often times, the templates produced by algorithms will require different data for different use cases. The enrollment type indicates what the use case for the created template will be, allowing implementors to specialize their templates if they so desire. The use cases supported by the API are:

Role	Description
Janice11Reference	The template will be used as a reference template for 1:1 verification.
Janice11Verification	The template will be used for verification against a reference template in 1:1 verification.
Janice1NProbe	The template will be used as a probe template in 1:N search.
Janice1NGallery	The template will be enrolled into a gallery and searched against in 1:N search.
JaniceCluster	The template will be used for clustering.

5.1.3 JaniceBatchPolicy

The JanICE API offers batch calls to accelerate computation for certain operations. For large batches, it is often advantageous to set an error handling policy to control if an application should fail immediately or flag and continue on an error. This is set explicity with the API batch policy. Possible policies are:

Policy	Description
JaniceAbortEarly	Stop processing immediately on an error
JaniceFlagAndFinish	Mark an error for the user but continue processing if possible

5.2 Structs

5.2.1 JaniceContext

A structure to hold hyperparameters. These hyperparameters may be set by the user to control execution of the implementation algorithms. Users should consult the relevant documentation for accepted values or ranges for these hyperparameters. Implementors should ensure the provided values are acceptable before using them.

Minimum Object Size

This function specifies a minimum object size as one of its parameters. This value indicates the minimum size of objects that the user would like to see detected. Often, increasing the minimum size can improve runtime of algorithms. The size is in pixels and corresponds to the length of the smaller side of the rectangle. This means a detection will be returned if and only if its smaller side is larger than the value specified. If the user does not wish to specify a minimum width 0 can be provided.

Hint

Clustering is generally considered to be an ill-defined problem, and most algorithms require some help determining the appropriate number of clusters. The hint parameter helps influence the number of clusters, though the implementation is free to ignore it. The goal of the hint is to provide user input for two use cases:

1. If the hint is between 0 - 1 it should be regarded as a purity requirement for the algorithm. A 1 indicates the user wants perfectly pure clusters, even if that means more clusters are returned. A 0 indicates that the user wants very few clusters returned and accepts there may be some errors.

2. If the hint is > 1 it represents an estimated upper bound on the number of object types in the set.

Fields

Name	Type	Description
policy	*JaniceDetectionPolicy*	The detection policy
min_object_size	uint32_t	The minumum object size of a detection. See *Minimum Object Size* for additional information
role	*JaniceEnrollmentType*	The enrollment type for a template
threshold	double	The minimum acceptable score for a search result.
max_returns	uint32_t	The maximum number of results a single search should return
hint	double	A hint to a clustering algorithm. See *Hint* for additional information
batch_policy	*JaniceBatchPolicy*	The batch policy

5.3 Functions

5.3.1 janice_init_default_context

Initialize hyperparameters of a context object with sensible defaults. The context object should be created by the user prior to calling this function.

Signature

```
JANICE_EXPORT JaniceError janice_init_default_context(JaniceContext* context);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
context	*JaniceContext**	The context to initialize. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

CHAPTER

SIX

TRAINING

6.1 Functions

6.1.1 janice_fine_tune

Fine tune an implementation using new data. This function can be used to adapt an algorithm to a new domain. It is optional and can return `JANICE_NOT_IMPLEMENTED`. Artifacts created from fine tuning should be stored on disk and will be loaded in a successive initialization of the API.

Signature

```
JANICE_EXPORT JaniceError janice_fine_tune(const JaniceMediaIterators* media,
                                           const JaniceDetectionsGroup* detections,
                                           int** labels,
                                           const char* output_prefix);
```

Thread Safety

This function is *Thread Unsafe*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	A list of media objects to fine tune with. After the function call, each iterator in the array will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
detections	const *JaniceDetectionsGroup**	A collection of location information for objects in the fine tuning data. There must be the same number of sublists in this structure as there are elements in `media`. The tracks in the `ith` sublist of this structure give locations in the `ith` media object.
labels	int**	A list of lists of labels for objects in the fine tuning data. `labels[i][j]` should give the label for the `jth` track in the `ith` sublist of `tracks`.
output_prefix	const char*	A path to an existing directory with write access for the application. After successful fine tuning, this directory should be populated with all files necessary to initialize the API. Future calls to the API can use the fine tuned algorithm by passing `output_prefix` as the `sdk_path` parameter in *janice_initialize*.

CHAPTER

SEVEN

DETECTION

7.1 Overview

In the context of this API, detection is used to refer to the identification of objects of interest within a *I/O* object. Detections are represented using the *JaniceDetectionType* object which implementors are free to define however they would like. For images, a detection is defined as a rectangle that bounds an object of interest and an associated confidence value. For video, a single object can exist in multiple frames. A rectangle and confidence are only relevant in a single frame. In this case, we define a detection as a list of (rectangle, confidence) pairs that track a single object through a video. It is not required that this list be dense however (i.e. frames can be skipped). To support this, we extend our representation of a detection to a (rectangle, confidence, frame) tuple where frame gives the index of the frame the rectangle was found in.

7.2 Structs

7.2.1 JaniceRect

A simple struct that represents a rectangle

Fields

Name	Type	Description
x	int	The x offset of the rectangle in pixels
y	int	The y offset of the rectangle in pixels
width	int	The width of the rectangle in pixels
height	int	The height of the rectangle in pixels

7.2.2 JaniceTrack

A structure to represent a track through a *JaniceMediaIterator* object. Tracks may be sparse (i.e. frames do not need to be sequential). Tracks are meant to follow a single object or area of interest, for example a face through multiple frames of a video.

Confidence

The confidence value indicates a likelihood that the rectangle actually bounds an object of interest. It is **NOT** required to be a probability and often only has meaning relative to other confidence values from the same algorithm. The only restriction is that a larger confidence value indicates a greater likelihood that the rectangle bounds an object.

Fields

Name	Type	Description
rects	*JaniceRect**	A list of rectangles surrounding areas of interest in the media. This list should be `length` elements.
confidences	float*	A confidence to associate with each rectangle in `rects`. See *Confidence* for details about confidence values in this API. This list should be `length` elements.
frames	uint32_t*	The frame indices associated with each rectangle in `rects`. A track may be sparse and the indicies in this list are required to be sequential. This list should be `length` elements.
length	size_t	The number of rectangles, confidences, and frames in this structure.

7.2.3 JaniceDetectionType

A struct that represents a detection. See *Detection* for more information.

7.2.4 JaniceDetection

A pointer to a *JaniceDetectionType* object.

Signature

```
typedef struct JaniceDetectionType* JaniceDetection;
```

7.2.5 JaniceDetections

A structure to represent a list of *JaniceDetection* objects.

Fields

Name	Type	Description
detections	*JaniceDetection**	An array of detection objects.
length	size_t	The number of elements in `detections`

7.2.6 JaniceDetectionsGroup

A structure to represent a list of *JaniceDetections* objects.

Fields

Name	Type	Description
group	*JaniceDetections*	An array of detections objects.
length	size_t	The number of elements in `group`

7.3 Callbacks

7.3.1 JaniceDetectionCallback

A function prototype to process *JaniceDetection* objects as they are found.

Signature

```
JaniceError (*JaniceDetectionCallback)(const JaniceDetection*, size_t, void*);
```

Thread Safety

This function is *Thread Unsafe*.

Parameters

Name	Type	Description
detection	const *JaniceDetection**	A detection object produced during the callback
index	size_t	The index of the media iterator in which the detection occured.
user_data	void*	User defined data that may assist in the processing of the detection. It is passed directly from the `\*_with_callback` function to the callback.

7.4 Functions

7.4.1 janice_create_detection_from_rect

Create a detection from a known rectangle. This is useful if a human has identified an object of interest and would like to run subsequent API functions on it. In the case where the input media is a video the given rectangle is considered an initial sighting of an object or region of interest. The implementation may detect additional sightings of the object in successive frames.

Signature

```
JANICE_EXPORT JaniceError janice_create_detection_from_rect(const␣
↪JaniceMediaIterator\* media,
                                                            const JaniceRect\* rect,
                                                            const uint32_t frame,
                                                            JaniceDetection*␣
↪detection);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterator**	A media object to create the detection from. After the function call, the iterator will exist in an undefined state. A user should call *reset* on the iterator before reusing it.
rect	const *JaniceRect**	A rectangle that bounds the object of interest.
frame	const uint32_t	An index to the frame in the media where the object of interest appears. If the media is an image this should be 0.
detection	*JaniceDetection**	An uninitialized pointer to a detection object. The object should allocated by the implementor during function execution. The user is responsible for freeing the object using *janice_free_detection*.

Example

```
JaniceMedia media; // Where media is a valid media object created previously

JaniceRect rect; // Create a bounding rectangle around an object of interest
rect.x      = 10; // The rectangle should fall within the bounds of the media
rect.y      = 10; // This code assumes media width > 110 and media height > 110
rect.width  = 100;
rect.height = 100;

JaniceDetection detection = NULL; // best practice to initialize to NULL
if (janice_create_detection(media, rect, 0 /* frame */, &detection) != JANICE_SUCCESS)
    // ERROR!
```

7.4.2 janice_create_detection_from_track

Create a detection from a known track. This is useful if a human has identified an object of interest and would like to run subsequent API functions on it.

Signature

```
JANICE_EXPORT JaniceError janice_create_detection_from_track(const␣
↪JaniceMediaIterator\* media,
                                                             const JaniceTrack\*␣
↪track,
                                                             JaniceDetection*␣
↪detection);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterator**	A media object to create the detection from. After the function call, the iterator will exist in an undefined state. A user should call *reset* on the iterator before reusing it.
track	const *JaniceTrack**	A track bounding a region of through 1 or more frames.
detection	*JaniceDetection**	An uninitialized pointer to a detection object. The object should be allocated by the implementor during function execution. The user is responsible for freeing the object by calling *janice_free_detection*.

7.4.3 janice_detect

Automatically detect objects in a media object. See *Detection* for an overview of detection in the context of this API.

Signature

```
JANICE_EXPORT JaniceError janice_detect(const JaniceMediaIterator* media,
                                        const JaniceContext* context,
                                        JaniceDetections* detections);
```

Thread Safety

This function is *Reentrant*.

Tracking

When the input media is a video, implementations may implement a form of object tracking to correlate multiple sightings of the same object into a single structure. There are a number of approaches and algorithms to implement object tracking. This API makes NO attempt to define or otherwise constrain how implementations handle tracking. Users should be warned that an implementation might output multiple tracks for a single object and that a single track might contain multiple objects in it by mistake. In some cases, which should be clearly documented in implementation documentation, it might be beneficial to perform a post-processing clustering step on the results tracks, which could help correlate multiple tracks of the same object.

Parameters

Name	Type	Description
media	const *JaniceMediaIterator**	A media object to run detection on. After the function call, the iterator will exist in an undefined state. A user should call *reset* on the iterator before reusing it.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
detections	*JaniceDetections**	A struct to hold the resulting detections. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_detections*

Example

```
JaniceContext context = nullptr;
if (janice_create_context(JaniceDetectAll, // detection policy
                          24, // min_object_size, only find objects where the smaller␣
↪side > 24 pixels
                          Janice1NProbe, // enrollment type, this shouldn't impact␣
↪detection
                          0, // threshold, this shouldn't impact detection
                          0, // max_returns, this shouldn't impact detection
                          0, // hint, this shouldn't impact detection
                          &context) != JANICE_SUCCESS)
    // ERROR!

JaniceMedia media; // Where media is a valid media object created previously
JaniceDetections detections;
if (janice_detect(media, context, &detections) != JANICE_SUCCESS)
    // ERROR!
```

7.4.4 janice_detect_with_callback

Run detection with a callback, which surfaces detections as they are made for processing. The callback accepts user data as input. It is important to remember that `JaniceMediaIterator` may be stateful and should not be part of the callback. The implementor is not responsible for ensuring that the state of `media` is not changed by the user during this call. The provided callback may return an error. If an error is returned by the callback, the implementation should abort and return that error as well. This function will always pass `0` to the index parameter of the callback.

Signature

```
JANICE_EXPORT JaniceError janice_detect_with_callback(const JaniceMediaIterator*␣
↪media,
                                                      const JaniceContext* context,
                                                      JaniceDetectionCallback␣
↪callback,
                                                      void* user_data);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const JaniceMediaIterator*	A media object to run detection on. After the function call, the iterator will exist in an undefined state. A user should call *reset* on the iterator before reusing it.
context	const JaniceContext*	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
callback	*JaniceDetectionCallback*	A pointer to a user defined callback function.
user_data	void*	A pointer to user defined data. This is passed to the callback function on each invocation.

7.4.5 janice_detect_batch

Detect faces in a batch of media objects. Batch processing can often be more efficient than serial processing, particularly if a GPU or co-processor is being utilized. This function reports per-image error codes. Depending on the batch policy given, it will return one of `JANICE_SUCCESS` if no errors occured, or `JANICE_BATCH_ABORTED_EARLY` or `JANICE_BATCH_FINISHED_WITH_ERRORS` if errors occured within the batch. In either case, any computation marked `JANICE_SUCCESS` in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_detect_batch(const JaniceMediaIterators* media,
                                              const JaniceContext* context,
                                              JaniceDetectionsGroup* detections,
                                              JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	An array of media iterators to run detection on. After the function call, each iterator in the array will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
detections	*JaniceDetectionsGroup**	A list of lists of detection objects. Each input media iterator can contain 0 or more possible detections. This output structure should mirror the input such that the sublist at index `i` should contain all of the detections found in media iterator `i`. If no detections are found in a particular media object an entry must still be present in the top-level output list and the sublist should have a length of 0. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_detections_group*
errors	*JaniceErrors**	A struct to hold per-image error codes. There must be the same number of errors as there are `media` unless the call aborted early, in which case there can be less. The `ith` error code should give the status of detection on the `ith` piece of media. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

7.4.6 janice_detect_batch_with_callback

Detect faces in a batch of media objects and surface detections as they are made for processing. Batch processing can often be more efficient than serial processing, particularly if a GPU or co-processor is being utilized. The callback accepts user data as input. It is important to remember that `JaniceMediaIterator` may be stateful and should not be part of the callback. The implementor is not responsible for ensuring that the state of `media` is not changed by the user during this call. The provided callback may return an error. If an error is returned by the callback, the implementation should abort and return that error as well.

Signature

```
JANICE_EXPORT JaniceError janice_detect_batch_with_callback(const␣
↪JaniceMediaIterators* media,
                                                            const JaniceContext*␣
↪context,
                                                            JaniceDetectionCallback␣
↪callback,
                                                            void* user_data);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	An array of media iterators to run detection on. After the function call, each iterator in the array will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
callback	*JaniceDetectionCallback*	A pointer to a user defined callback function.
user_data	void*	A pointer to user defined data. This is passed to the callback function on each invocation.

7.4.7 janice_detection_get_track

Get a track object from a detection. The returned track should contain all rectangles, confidences, and frame indicies stored in the detection.

Signature

```
JANICE_EXPORT JaniceError janice_detection_get_track(const JaniceDetection detection,
                                                     JaniceTrack* track);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
detection	const *JaniceDetection*	The detection to get the track from.
track	*JaniceTrack**	The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for free this object by calling *janice_clear_track*.

7.4.8 janice_detection_get_attribute

Get an attribute from a detection. Attributes are additional metadata that an implementation might have when creating a detection. Examples from face detection include gender, ethnicity, and / or landmark locations. Implementors are responsible for providing documentation on any attributes they support, valid key values and possible return values.

Signature

```
JANICE_EXPORT JaniceError janice_detection_get_attribute(const JaniceDetection␣
↪detection,
                                                         const char* key,
                                                         char** value);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
detection	const *JaniceDetection*	The detection object to extract the attribute from.
key	const char*	A null-terminated key to look up a specific attribute. Valid keys must be defined and documented by the implementor.
value	char**	An uninitialized char* to hold the value of the attribute. This object should be allocated by the implementor during the function call. This object must be null-terminated. The user is responsible for the object by calling *janice_free_attribute*.

7.4.9 janice_serialize_detection

Serialize a *JaniceDetection* object to a flat buffer.

Signature

```
JANICE_EXPORT JaniceError janice_serialize_detection(const JaniceDetection detection,
                                                     uint8_t** data,
                                                     size_t* len);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
detection	const *JaniceDetection*	A detection object to serialize
data	uint8_t**	An uninitialized buffer to hold the flattened data. The implementor should allocate this object during the function call. The user is required to free the object with *janice_free_buffer*.
len	size_t*	The length of the flat buffer after it is filled. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

Example

```
JaniceDetection detection; // Where detection is a valid detection created
                           // previously.

uint8_t* buffer = NULL;
size_t buffer_len;
janice_serialize_detection(detection, &buffer, &buffer_len);
```

7.4.10 janice_deserialize_detection

Deserialize a *JaniceDetection* object from a flat buffer.

Signature

```
JANICE_EXPORT JaniceError janice_deserialize_detection(const uint8_t* data,
                                                       size_t len,
                                                       JaniceDetection* detection);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
data	const uint8_t*	A buffer containing data from a flattened detection object.
len	size_t	The length of the flat buffer.
detection	*JaniceDetection**	An uninitialized detection object. This object should be allocated by the implementor during the function call. Users are required to free the object with *janice_free_detection*.

Example

```
const size_t buffer_len = K; // Where K is the known length of the buffer
uint8_t buffer[buffer_len];

FILE* file = fopen("serialized.detection", "r");
fread(buffer, 1, buffer_len, file);

JaniceDetection detection = nullptr;
janice_deserialize_detection(buffer, buffer_len, detection);

fclose(file);
```

7.4.11 janice_read_detection

Read a detection from a file on disk. This method is functionally equivalent to the following-

```
const size_t buffer_len = K; // Where K is the known length of the buffer
uint8_t buffer[buffer_len];

FILE* file = fopen("serialized.detection", "r");
fread(buffer, 1, buffer_len, file);

JaniceDetection detection = nullptr;
janice_deserialize_detection(buffer, buffer_len, detection);

fclose(file);
```

It is provided for memory efficiency and ease of use when reading from disk.

Signature

```
JANICE_EXPORT JaniceError janice_read_detection(const char* filename,
                                                JaniceDetection* detection);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
filename	const char*	The path to a file on disk
detection	*JaniceDetection**	An uninitialized detection object.

Example

```
JaniceDetection detection = NULL;
if (janice_read_detection("example.detection", &detection) != JANICE_SUCCESS)
    // ERROR!
```

7.4.12 janice_write_detection

Write a detection to a file on disk. This method is functionally equivalent to the following-

```
JaniceDetection detection; // Where detection is a valid detection created
                           // previously.

uint8_t* buffer = NULL;
size_t buffer_len;
janice_serialize_detection(detection, &buffer, &buffer_len);

FILE* file = fopen("serialized.detection", "w+");
fwrite(buffer, 1, buffer_len, file);

fclose(file);
```

It is provided for memory efficiency and ease of use when writing to disk.

Signature

```
JANICE_EXPORT JaniceError janice_write_detection(const JaniceDetection detection,
                                                 const char* filename);
```

ThreadSafety

This function is *Reentrant*.

Parameters

Name	Type	Description
detection	const *JaniceDetection*	The detection object to write to disk.
filename	const char*	The path to a file on disk

Example

```
JaniceDetection detection; // Where detection is a valid detection created
                           // previously
if (janice_write_detection(detection, "example.detection") != JANICE_SUCCESS)
    // ERROR!
```

7.4.13 janice_free_buffer

Release the memory for an allocated buffer.

Signature

```
JANICE_EXPORT JaniceError janice_free_bufferuint8_t** buffer);
```

Thread Safety

This function is *Reentrant*

Parameters

Name	Type	Description
buffer	uint8_t**	The buffer to free

7.4.14 janice_free_detection

Free any memory associated with a *JaniceDetection* object.

Signature

```
JANICE_EXPORT JaniceError janice_free_detection(JaniceDetection* detection);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
detection	*JaniceDetection**	A detection object to free.

Example

```
JaniceDetection detection; // Where detection is a valid detection object
                           // created previously
if (janice_free_detection(&detection) != JANICE_SUCCESS)
    // ERROR!
```

7.4.15 janice_clear_detections

Free any memory associated with a *JaniceDetections* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_detections(JaniceDetections* detections);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
detections	*JaniceDetections* *	A detection object to clear.

7.4.16 janice_clear_detections_group

Free any memory associated with a *JaniceDetectionsGroup* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_detections_group(JaniceDetectionsGroup\*␣
↪group);
```

7.4.17 janice_clear_track

Free any memory associated with a *JaniceTrack* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_track(JaniceTrack* track);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
track	*JaniceTrack**	The track object to clear.

7.4.18 janice_free_attribute

Free any memory associated with an attribute value.

Signature

```
JANICE_EXPORT JaniceError janice_free_attribute(char** value);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
attribute	char**	The attribute to free.

CHAPTER

EIGHT

ENROLLMENT

8.1 Overview

This API defines feature extraction as the process of turning 1 or more *Detection* API objects that refer to the same object of interest into a single representation. This representation is defined in the API using the *JaniceTemplateType* object. In some cases (e.g. face recognition) this model of [multiple detections] -> [single representation] contradicts the current paradigm of [single detection] -> [single representation]. Implementors are free to implement whatever paradigm they choose internally (i.e. a JanICE template could be a simple list of single detection templates) provided the *Comparison / Search* functions work appropriately.

8.1.1 Failure To Enroll

For computer vision use cases, it is common to implement quality checks that can cause a template to fail during enrollment if it is missing certain characteristics. In this API templates should fail to enroll (FTE) quietly. This means that successive operations using an FTE template should still work without error. For example, calling *janice_verify* with an FTE template and a successful template should still return a score, even if that score is a predetermined constant value like `-FLOAT_MAX`. Users can query a template to see if it failed to enroll using the *janice_template_is_fte* function and may choose to manually discard it if they desire.

8.2 Enumerations

8.2.1 JaniceFeatureVectorType

The data type of the feature vector returned by *janice_template_get_feature_vector*. Supported data types are:

Data Type	Description
JaniceInt8	8 bit signed integer. The associated *C* type is `int8_t`
JaniceInt16	16 bit signed integer. The associated *C* type is `int16_t`
JaniceInt32	32 bit signed integer. The associated *C* type is `int32_t`
JaniceInt64	64 bit signed integer. The associated *C* type is `int64_t`
JaniceUInt8	8 bit unsigned integer. The associated *C* type is `uint8_t`
JaniceUInt16	16 bit unsigned integer. The associated *C* type is `uint16_t`
JaniceUInt32	32 bit unsigned integer. The associated *C* type is `uint32_t`
JaniceUInt64	64 bit unsigned integer. The associated *C* type is `uint64_t`
JaniceFloat	32 bit floating point number. The associated *C* type is `float`
JaniceDouble	64 bit floating point number. The associated *C* type is `double`

8.3 Structs

8.3.1 JaniceTemplateType

A struct that represents a template.

8.4 Typedefs

8.4.1 JaniceTemplate

A pointer to a *JaniceTemplateType* object.

Signature

```
typedef struct JaniceTemplateType* JaniceTemplate;
```

8.4.2 JaniceTemplates

A structure representing a list of *JaniceTemplate* objects.

Fields

Name	Type	Description
tmpls	*JaniceTemplate**	An array of template objects.
length	size_t	The number of elements in `tmpls`.

8.4.3 JaniceTemplatesGroup

A structure representing a list of *JaniceTemplates* objects.

Fields

Name	Type	Description
group	*JaniceTemplates**	An array of templates objects.
length	size_t	The number of elements in `group`.

8.5 Callbacks

8.5.1 JaniceEnrollMediaCallback

A function prototype to process *JaniceTemplate* and *JaniceDetection* objects as they are found.

Signature

```
JaniceError (*JaniceEnrollMediaCallback)(const JaniceTemplate*, const␣
↪JaniceDetection*, size_t, void*);
```

Thread Safety

This function is *Thread Unsafe*.

Parameters

Name	Type	Description
tmpl	const *JaniceTemplate**	A template object enrolled during the function
detection	const *JaniceDetection**	A detection object containing the location of the enrolled template
index	size_t	The index of the media iterator the template was enrolled from.
user_data	void*	User defined data that may assist in the processing of template. It is passed directly from the `\*_with_callback` function to the callback.

8.5.2 JaniceEnrollDetectionsCallback

A function prototype to process *JaniceTemplate* objects as they are created.

Signature

```
JaniceError (*JaniceEnrollDetectionsCallback)(const JaniceTemplate*, size_t, void*);
```

Thread Safety

This function is *Thread Unsafe*.

Parameters

Name	Type	Description
tmpl	const *JaniceTemplate**	A template object enrolled during the function
index	size_t	The index of the media iterator group the template was enrolled from.
user_data	void*	User defined data that may assist in the processing of the detection. It is passed directly from the `\*_with_callback` function to the callback.

8.6 Functions

8.6.1 janice_enroll_from_media

Detect and enroll templates from a single media file. Detection should respect the provided minimum object size and detection policy. This function may produce 0 or more templates, depending on the number of objects found in the media.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_media(const JaniceMediaIterator* media,
                                                    const JaniceContext* context,
                                                    JaniceTemplates* tmpls,
                                                    JaniceDetections* detections);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterator**	The media to detect and enroll templates from. After the function call, the iterator will exist in an undefined state. A user should call *reset* on the iterator before reusing it.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
tmpls	*JaniceTemplates**	A struct to hold the templates enrolled from the media. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear this object by calling *janice_clear_templates*
detections	*JaniceDetections**	A struct to hold the detection information for each of the templates enrolled from the media. This object should have the same number of elements as `tmpls`. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear this object by calling *janice_clear_detections*.

8.6.2 janice_enroll_from_media_with_callback

Run detection with a callback, which surfaces detections as they are made for processing. The callback accepts user data as input. It is important to remember that `JaniceMediaIterator` may be stateful and should not be part of the callback. The implementor is not responsible for ensuring that the state of `media` is not changed by the user during this call. The provided callback may return an error. If an error is returned by the callback, the implementation should abort and return that error as well. This function will always pass `0` to the index parameter of the callback.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_media_with_callback(const␣
↪JaniceMediaIterator* media,
                                                                 const JaniceContext*␣
↪context,
                                                                ␣
↪JaniceEnrollMediaCallback callback,
                                                                 void* user_data);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterator**	A media object to run detection and enrollment on. After the function call, the iterator will exist in an undefined state. A user should call *reset* on the iterator before reusing it.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
callback	*JaniceEnrollMediaCallback*	A pointer to a user defined callback function.
user_data	void*	A pointer to user defined data. This is passed to the callback function on each invocation.

8.6.3 janice_enroll_from_media_batch

Detect and enroll templates from a batch of media objects. Batch processing can often be more efficient then serial processing of a collection of data, particularly if a GPU or co-processor is being utilized. This function reports per-image error codes. Depending on the batch policy given, it will return one of `JANICE_SUCCESS` if no errors occured, or `JANICE_BATCH_ABORTED_EARLY` or `JANICE_BATCH_FINISHED_WITH_ERRORS` if errors occured within the batch. In either case, any computation marked `JANICE_SUCCESS` in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_media_batch(const JaniceMediaIterators*␣
↪media,
                                                         const JaniceContext* context,
                                                         JaniceTemplatesGroup* tmpls,
                                                         JaniceDetectionsGroup*␣
↪detections,
                                                         JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	An array of media iterators to enroll. After the function call, each iterator in the array will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
tmpls	*JaniceTemplatesGroup**	A list of lists of template objects. Each input media iterator can contain 0 or more possible templates. This output structure should mirror the input such that the sublist at index `i` should contain all of the templates enrolled from media iterator `i`. If no templates are enrolled from a particular media object an entry must still be present in the top-level output list and the sublist should have a length of 0. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_templates_group*.
detections	*JaniceDetectionsGroup**	A list of lists of track objects. The top level list should have the same number of elements as `tmpls` and sublist `i` should have the same number of elements as `tmpls` sublist i. Each track in the sublist should provide the location information for where the corresponding template was enrolled from. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_detections_group*.
errors	*JaniceErrors**	A struct to hold per-image error codes. There must be the same number of errors as there are `media` unless the call aborted early, in which case there can be less. The `ith` error code should give the status of detection on the `ith` piece of media. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

8.6.4 janice_enroll_from_media_batch_with_callback

Run batched detection and enrollment with a callback, which surfaces templates and associated detections they are made for processing. Batch processing can often be more efficient than serial processing, particularly if a GPU or co-processor is being utilized. The callback accepts user data as input. It is important to remember that `JaniceMediaIterator` may be stateful and should not be part of the callback. The implementor is not responsible for ensuring that the state of `media` is not changed by the user during this call. The provided callback may return an error. If an error is returned by the callback, the implementation should abort and return that error as well.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_media_batch_with_callback(const␣
↪JaniceMediaIterators* media,
                                                                        const␣
↪JaniceContext* context,
```

(continues on next page)

(continued from previous page)

```
                                                                              ␣
↪JaniceEnrollMediaCallback callback,
                                                                               void* user_
↪data);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	A list of media objects to run detection and enrollment on. After the function call, each iterator will exist in an undefined state. A user should call *reset* on each iterator before reusing it.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
callback	*JaniceEnrollMediaCallback*	A pointer to a user defined callback function.
user_data	void*	A pointer to user defined data. This is passed to the callback function on each invocation.

8.6.5 janice_enroll_from_detections

Create a *JaniceTemplate* object from an array of detections.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_detections(const JaniceMediaIterators*␣
↪media,
                                                      const JaniceDetections*␣
↪detections,
                                                      const JaniceContext* context,
                                                      JaniceTemplate* tmpl);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	An array of media objects. The array should have the same length as `detections`. After the function call, each iterator in the array will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
detections	const *JaniceDetections**	An array of detection objects. Each detection in the array should represent a unique sighting of the same object. The `ith` detection in the array represents a sighting in the `ith` element in `media`. This array should have the same length as `media`.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
tmpl	*JaniceTemplate**	An uninitialized template object. The implementor should allocate this object during the function call. The user is responsible for freeing the object by calling *janice_free_template*.

8.6.6 janice_enroll_from_detections_batch

Create a set of *JaniceTemplate* objects from an array of detections. Batch processing can often be more efficient then serial processing of a collection of data, particularly if a GPU or co-processor is being utilized. This function reports per media error codes. Depending on the batch policy given, it will return one of `JANICE_SUCCESS` if no errors occured, or `JANICE_BATCH_ABORTED_EARLY` or `JANICE_BATCH_FINISHED_WITH_ERRORS` if errors occured within the batch. In either case, any computation marked `JANICE_SUCCESS` in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_detections_batch(const␣
↪JaniceMediaIteratorsGroup* media,
                                                              const␣
↪JaniceDetectionsGroup* detections,
                                                              const JaniceContext*␣
↪context,
                                                              JaniceTemplates* tmpls,
                                                              JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIteratorsGroup**	A list of lists of media objects. Each sublist in this object should contain all of the media corresponding to unique sightings of an object of interest. The `ith` sublist should be the same length at the `ith` sublist of `detections`. The number of sublists should match the number of sublists in `detections`. After the function call, each iterator in each sublist of the group will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
detections	const *JaniceDetectionsGroup**	A list of lists of detection objects. Multiple detections can be enrolled into a single template, for example if detections correspond to multiple views of the object of interest. Each sublist in this object should contain all detections that should be enrolled into a single template. The `jth` element in the `ith` sublist should represent a sighting in the `jth` element in the `ith` sublist of `media`.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
tmpls	*JaniceTemplates**	A structure to hold the enrolled templates. This should have the same number of elements as `detections`. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_templates*.
errors	*JaniceErrors**	A struct to hold per-image error codes. There must be the same number of errors as there are `media` groups unless the call aborted early, in which case there can be less. The `ith` error code should give the status of enrollment on the `ith` group of media. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

8.6.7 janice_enroll_from_detections_batch_with_callback

Create templates from a batch of sightings. Batch processing can often be more efficient than serial processing, particularly if a GPU or co-processor is being utilized. The callback accepts user data as input. It is important to remember that `JaniceMediaIterator` may be stateful and should not be part of the callback. The implementor is not responsible for ensuring that the state of `media` is not changed by the user during this call. The provided callback may return an error. If an error is returned by the callback, the implementation should abort and return that error as well.

Signature

```
JANICE_EXPORT JaniceError janice_enroll_from_detections_batch_with_callback(const␣
↪JaniceMediaIteratorsGroup* media,
                                                                            const␣
↪JaniceDetectionsGroup* detections,
                                                                            const␣
↪JaniceContext* context,
                                                                          ␣
↪JaniceEnrollDetectionsCallback callback,
```

(continues on next page)

(continued from previous page)

```
                                                                   void*␣
↪user_data);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIteratorsGroup**	A list of lists of media objects. Each sublist in this object should contain all of the media corresponding to unique sightings of an object of interest. The `ith` sublist should be the same length at the `ith` sublist of `detections`. The number of sublists should match the number of sublists in `detections`. After the function call, each iterator in each sublist of the group will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
detections	const *JaniceDetectionsGroup**	A list of lists of detection objects. Multiple detections can be enrolled into a single template, for example if detections correspond to multiple views of the object of interest. Each sublist in this object should contain all detections that should be enrolled into a single template. The `jth` element in the `ith` sublist should represent a sighting in the `jth` element in the `ith` sublist of `media`.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
callback	*JaniceEnrollDetectionsCallback*	A pointer to a user defined callback function.
user_data	void*	A pointer to user defined data. This is passed to the callback function on each invocation.

8.6.8 janice_template_is_fte

Query to see if a template has failed to enroll. See *Failure To Enroll* for additional information.

Signature

```
JANICE_EXPORT JaniceError janice_template_is_fte(const JaniceTemplate tmpl,
                                                  int* fte);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpl	const *JaniceTemplate*	The template object to query.
fte	int*	FTE flag. If the template has not failed to enroll this should equal 0. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

8.6.9 janice_template_get_attribute

Get a metadata value from a template using a key string. The valid set of keys is determined by the implementation and must be included in their delivered documentation. The possible return values for a valid key are also implementation specific. Invalid keys should return an error.

Signature

```
JANICE_EXPORT JaniceError janice_template_get_attribute(const JaniceTemplate tmpl,
                                                        const char* key,
                                                        char** value);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpl	const *JaniceTemplate*	A template object to query the attribute from.
key	const char*	A null-terminated key to look up a specific attribute. Valid keys must be defined and documented by the implementor.
value	char**	An uninitialized char* to hold the value of the attribute. This object should be allocated by the implementor during the function call. This object must be null-terminated. The user is responsible for the object by calling *janice_free_attribute*.

8.6.10 janice_template_get_feature_vector

Extract a feature vector from a template. The requirements of the feature vector are still being defined.

Signature

```
JANICE_EXPORT JaniceError janice_template_get_feature_vector(const JaniceTemplate␣
↪tmpl,
                                                             const␣
↪JaniceFeatureVectorType feature_vector_type,
                                                             void** feature_vector,
                                                             size_t* length);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpl	const *JaniceTemplate*	A template object to query the feature vector from.
feature_vector_type	const *JaniceFeatureVectorType*	The data type of the returned feature vector. It should be possible to interpret `feature_vector` as a `size` length array of the feature vector type.
feature_vector	void**	A one-dimensional array containing the feature vector data. The user is responsible for the object by calling *janice_free_feature_vector*.
size	size_t*	The length of `feature_vector`.

8.6.11 janice_serialize_template

Serialize a *JaniceTemplate* object to a flat buffer.

Signature

```
JANICE_EXPORT JaniceError janice_serialize_template(const JaniceTemplate tmpl,
                                                    uint8_t** data,
                                                    size_t* len);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpl	const *JaniceTemplate*	A template object to serialize
data	uint8_t**	An uninitialized buffer to hold the flattened data. The implementor should allocate this object during the function call. The user is responsible for freeing the object by calling *janice_free_buffer*
len	size_t*	The length of the flat buffer. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

Example

```
JaniceTemplate tmpl; // Where tmpl is a valid template created
                     // previously.

JaniceBuffer buffer = NULL;
size_t buffer_len;
janice_serialize_template(tmpl, &buffer, &buffer_len);
```

8.6.12 janice_deserialize_template

Deserialize a *JaniceTemplate* object from a flat buffer.

Signature

```
JANICE_EXPORT JaniceError janice_deserialize_template(const uint8_t* data,
                                                      size_t len,
                                                      JaniceTemplate* tmpl);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
data	const uint8_t*	A buffer containing data from a flattened template object.
len	size_t	The length of the flat buffer.
tmpl	*JaniceTemplate**	An uninitialized template object. The implementor should allocate this object during the function call. The user is responsible for freeing the object by calling *janice_free_template*.

Example

```
const size_t buffer_len = K; // Where K is the known length of the buffer
uint8_t buffer[buffer_len];

FILE* file = fopen("serialized.template", "r");
fread(buffer, 1, buffer_len, file);

JaniceTemplate tmpl = NULL; // best practice to initialize to NULL
janice_deserialize_template(buffer, buffer_len, tmpl);

fclose(file);
```

8.6.13 janice_read_template

Read a template from a file on disk. This method is functionally equivalent to the following-

```
const size_t buffer_len = K; // Where K is the known length of the buffer
uint8_t buffer[buffer_len];

FILE* file = fopen("serialized.template", "r");
fread(buffer, 1, buffer_len, file);

JaniceTemplate tmpl = nullptr;
janice_deserialize_template(buffer, buffer_len, tmpl);

fclose(file);
```

It is provided for memory efficiency and ease of use when reading from disk.

Signature

```
JANICE_EXPORT JaniceError janice_read_template(const char* filename,
                                               JaniceTemplate* tmpl);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
filename	const char*	The path to a file on disk
tmpl	*JaniceTemplate**	An uninitialized template object. The implementor should allocate this object during the function call. The user is responsible for freeing the object by calling *janice_free_template*.

Example

```
JaniceTemplate tmpl = NULL;
if (janice_read_template("example.template", &tmpl) != JANICE_SUCCESS)
    // ERROR!
```

8.6.14 janice_write_template

Write a template to a file on disk. This method is functionally equivalent to the following-

```
JaniceTemplate tmpl; // Where tmpl is a valid template created
                     // previously.

JaniceBuffer buffer = NULL;
size_t buffer_len;
janice_serialize_template(tmpl, &buffer, &buffer_len);

FILE* file = fopen("serialized.template", "w+");
fwrite(buffer, 1, buffer_len, file);

fclose(file);
```

It is provided for memory efficiency and ease of use when writing to disk.

Signature

```
JANICE_EXPORT JaniceError janice_write_template(JaniceTemplate tmpl,
                                                const char* filename);
```

ThreadSafety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpl	*JaniceTemplate*	The template object to write to disk.
filename	const char*	The path to a file on disk.

Example

```
JaniceTemplate tmpl; // Where tmpl is a valid template created
                     // previously
if (janice_write_template(tmpl, "example.template") != JANICE_SUCCESS)
    // ERROR!
```

8.6.15 janice_free_template

Free any memory associated with a *JaniceTemplate* object.

Signature

```
JANICE_EXPORT JaniceError janice_free_template(JaniceTemplate* tmpl);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpl	*JaniceTemplate*	A template object to free.

Example

```
JaniceTemplate tmpl; // Where tmpl is a valid template object created previously
if (janice_free_template(&tmpl) != JANICE_SUCCESS)
    // ERROR!
```

8.6.16 janice_clear_templates

Free any memory associated with a *JaniceTemplates* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_templates(JaniceTemplates* templates);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpls	*JaniceTemplates**	A templates objects to clear.

8.6.17 janice_clear_templates_group

Free any memory associated with a *JaniceTemplatesGroup* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_templates_group(JaniceTemplatesGroup* group);
```

Parameters

Name	Type	Description
group	*JaniceTemplatesGroup**	A templates group to clear.

8.6.18 janice_free_feature_vector

Free a feature vector returned by *janice_template_get_feature_vector*

Signature

```
JANICE_EXPORT JaniceError janice_free_feature_vector(void** feature_vector);
```

Parameters

Name	Type	Description
feature_vector	void**	A feature vector to free.

CHAPTER

NINE

GALLERY

9.1 Overview

This API defines a gallery object that represents a collection of templates. Galleries are useful in the 1-N use case (see *Comparison / Search*) when a user would like to query an unknown probe template against a set of known identities. A naive implementation of a gallery might be a simple array of templates. Often however, implementations have optimized algorithms or data structures that can lead to more efficient search times. It is recommended that advanced data structures be implemented as part of a gallery. Please note however the rules on gallery modification:

1. Gallery objects may be modified (templates inserted or removed) at any time.
2. It is understood that some preprocessing might need to be done between gallery modification and efficient search. A function *janice_gallery_prepare* exists for this purpose. The calling of this function is **OPTIONAL**. Please see *janice_gallery_prepare* for more information.

9.2 Structs

9.2.1 JaniceGalleryType

A struct that represents a gallery.

9.3 Typedefs

9.3.1 JaniceGallery

A pointer to a *JaniceGalleryType* object.

Signature

```
typedef struct JaniceGalleryType* JaniceGallery;
```

9.3.2 JaniceTemplateIds

A structure representing a list of unique template ids.

Fields

Name	Type	Description
ids	uint64_t*	An array of template id objects
length	size_t	The number of elements in `ids`

9.3.3 JaniceTemplateIdsGroup

A structure representing a list of *JaniceTemplateIds* objects.

Fields

Name	Type	Description
group	*JaniceTemplateIds**	An array of template ids objects.
length	size_t	The number of elements in `group`

9.4 Functions

9.4.1 janice_create_gallery

Create a *JaniceGallery* object from a list of templates and unique ids.

Signature

```
JANICE_EXPORT JaniceError janice_create_gallery(const JaniceTemplates* tmpls,
                                                const JaniceTemplateIds* ids,
                                                JaniceGallery* gallery);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpls	const *JaniceTemplates**	An array of templates to add to the gallery. This can be `NULL` which would create an empty gallery. Data should be copied into the gallery. It is valid to pass an array with length 0 into this function, in which case an empty gallery should be initialized. This structure must have the same number of elements as `ids`.
ids	const *JaniceTemplateIds**	A set of unique indentifiers to associate with the templates in `tmpls`. The `ith` id in this array corresponds to the `ith` input template. This structure must have the same number of elements as `tmpls`.
gallery	*JaniceGallery**	An uninitialized gallery object. The implementor should allocate this object during the function call. The user is required to free this object by calling *janice_free_gallery*.

Example

```
JaniceTemplates tmpls; // Where tmpls is a valid array of valid template
                       // objects created previously
JaniceTemplateIds ids; // Where ids is a valid array of unique unsigned integers that
                       // is the same length as tmpls
JaniceGallery gallery = NULL; // best practice to initialize to NULL

if (janice_create_gallery(&tmpls, &ids, &gallery) != JANICE_SUCCESS)
    // ERROR!
```

9.4.2 janice_gallery_reserve

Reserve space in a gallery for N templates. This can save repeated allocations when doing multiple iterative inserts.

Signature

```
JANICE_EXPORT JaniceError janice_gallery_reserve(JaniceGallery gallery,
                                                 size_t n);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	The gallery to reserve space in.
n	size_t	The number of templates to reserve space for.

9.4.3 janice_gallery_insert

Insert a template into a gallery object. The template data should be copied into the gallery as the template may be deleted after this function.

Signature

```
JANICE_EXPORT JaniceError janice_gallery_insert(JaniceGallery gallery,
                                                const JaniceTemplate tmpl,
                                                const uint64_t id);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	A gallery object to insert the template into.
tmpl	const *JaniceTemplate*	A template object to insert into the gallery. The template was created with the `Janice1NGallery` role. The template should be copied into the gallery. This object must remain in a valid state after this function call.
id	const uint64_t	A unique id to associate with the input template. If the id is not unique the implementor should return `JANICE_DUPLICATE_ID`.

Example

```
JaniceTemplate tmpl; // Where tmpl is a valid template object created
                     // previously
uint64_t id; // Where id is a unique integer to associate with tmpl. This
                     // integer should not exist in the gallery
JaniceGallery gallery; // Where gallery is a valid gallery object created
                       // previously

if (janice_gallery_insert(gallery, tmpl, id) != JANICE_SUCCESS)
    // ERROR!
```

9.4.4 janice_gallery_insert_batch

Insert a batch of templates into a gallery. Batch processing can often be more efficient then serial processing of a collection of data, particularly if a GPU or co-processor is being utilized. This function reports per template error codes. Depending on the batch policy given, it will return one of `JANICE_SUCCESS` if no errors occured, or `JANICE_BATCH_ABORTED_EARLY` or `JANICE_BATCH_FINISHED_WITH_ERRORS` if errors occured within the batch. In either case, any computation marked `JANICE_SUCCESS` in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_gallery_insert_batch(JaniceGallery gallery,
                                                      const JaniceTemplates* tmpls,
                                                      const JaniceTemplateIds* ids,
                                                      const JaniceContext* context,
                                                      JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	The gallery to insert the templates into.
tmpls	const *JaniceTemplates**	The array of templates to insert in to the gallery. Each template was created with the `Janice1NGallery` role. Each template should be copied into the gallery by the implementor and must remain in a valid state after this function call. This structure must have the same number of elements as `ids`.
ids	const *JaniceTemplateIds**	The array of unique ids to associate with `tmpls`. The `ith` id in this structure corresponds to the `ith` template in `tmpls`. This structure must have the same number of elements as `tmpls`.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
errors	*JaniceErrors**	A struct to hold per-template error codes. There must be the same number of errors as there are `tmpls` unless the call aborted early, in which case there can be less. The `ith` error code should give the status of insertion on the `ith` template. The user is responsible for allocating memory for the struct before the function call. The implementor is responsibile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

9.4.5 janice_gallery_remove

Remove a template from a gallery object using its unique id.

Signature

```
JANICE_EXPORT JaniceError janice_gallery_remove(JaniceGallery gallery,
                                                const uint64_t id);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	The gallery object to remove a template from.
id	const uint64_t	The unique identifier for the template to remove from the gallery. If no template with the given ID is found in the gallery this function should return `JANICE_MISSING_ID`.

Example

```
JaniceTemplate tmpl; // Where tmpl is a valid template object created
                     // previously
uint64_t id = 0; // A unique integer id to associate with tmpl.

JaniceGallery gallery; // Where gallery is a valid gallery object created
                       // previously that does not have a template with id '0'
                       // already inserted in it.

// Insert the template with id 0
if (janice_gallery_insert(gallery, tmpl, id) != JANICE_SUCCESS)
    // ERROR!

// Now we can remove the template
if (janice_gallery_remove(gallery, id) != JANICE_SUCCESS)
    // ERROR!
```

9.4.6 janice_gallery_remove_batch

Remove a batch of templates from a gallery. Batch processing can often be more efficient then serial processing of a collection of data, particularly if a GPU or co-processor is being utilized. This function reports per template error codes. Depending on the batch policy given, it will return one of `JANICE_SUCCESS` if no errors occured, or `JANICE_BATCH_ABORTED_EARLY` or `JANICE_BATCH_FINISHED_WITH_ERRORS` if errors occured within the batch. In either case, any computation marked `JANICE_SUCCESS` in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_gallery_remove_batch(JaniceGallery gallery,
                                                      const JaniceTemplateIds* ids,
                                                      const JaniceContext* context,
                                                      JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	The gallery object to remove the templates from.
ids	const *JaniceTemplateIds**	The unique identifiers for the templates to remove from the gallery.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
errors	*JaniceErrors**	A struct to hold per-id error codes. There must be the same number of errors as there are `ids` unless the call aborted early, in which case there can be less. The `ith` error code should give the status of removal on the `ith` id. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

9.4.7 janice_gallery_prepare

Prepare a gallery for search. Implementors can use this function as an opportunity to streamline gallery objects to accelerate the search process. The calling convention for this function is **NOT** specified by the API, this means that this function is not guaranteed to be called before *janice_search*. It also means that templates can be added to a gallery before and after this function is called. Implementations should handle all of these calling conventions. However, users should be aware that this function may be computationally expensive. They should strive to call it only at critical junctions before search and as few times as possible overall.

Signature

```
JANICE_EXPORT JaniceError janice_gallery_prepare(JaniceGallery gallery);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	A gallery object to prepare

Example

```
JaniceTemplate* tmpls; // Where tmpls is a valid array of valid template
                       // objects created previously
JaniceTemplateIds ids; // Where ids is a valid array of unique unsigned integers that
                       // is the same length as tmpls
JaniceTemplate tmpl; // Where tmpl is a valid template object created
                     // previously
JaniceTemplateId id; // Where id is a unique integer id to associate with tmpl.

JaniceGallery gallery = NULL; // best practice to initialize to NULL

if (janice_create_gallery(tmpls, ids, &gallery) != JANICE_SUCCESS)
    // ERROR!

// It is valid to run search without calling prepare
if (janice_search(tmpl, gallery ... ) != JANICE_SUCCESS)
    // ERROR!

// Prepare can be called after search
if (janice_gallery_prepare(gallery) != JANICE_SUCCESS)
    // ERROR!

// Search can be called again right after prepare
if (janice_search(tmpl, gallery ... ) != JANICE_SUCCESS)
    // ERROR!

// Insert another template into the gallery. This is valid after the gallery
// has been prepared
if (janice_gallery_insert(gallery, tmpl, 112) != JANICE_SUCCESS)
    // ERROR!

// Prepare the gallery again
if (janice_gallery_prepare(gallery) != JANICE_SUCCESS)
    // ERROR!
```

9.4.8 janice_serialize_gallery

Serialize a *JaniceGallery* object to a flat buffer.

Signature

```
JANICE_EXPORT JaniceError janice_serialize_gallery(const JaniceGallery gallery,
                                                   uint8_t** data,
                                                   size_t* len);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	A gallery object to serialize
data	uint8_t**	An uninitialized buffer to hold the flattened data. The implementor allocate this object during the function call. The user is responsible for freeing this object by calling *janice_free_buffer*.
len	size_t*	The length of the flat buffer after it is allocated. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.

Example

```
JaniceGallery gallery; // Where gallery is a valid gallery created
                       // previously.

uint8_t\* buffer = NULL;
size_t buffer_len;
janice_serialize_gallery(gallery, &buffer, &buffer_len);
```

9.4.9 janice_deserialize_gallery

Deserialize a *JaniceGallery* object from a flat buffer.

Signature

```
JANICE_EXPORT JaniceError janice_deserialize_gallery(const uint8_t* data,
                                                     size_t len,
                                                     JaniceGallery* gallery);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
data	const uint8_t*	A buffer containing data from a flattened gallery object.
len	size_t	The length of the flat buffer.
gallery	*JaniceGallery**	An uninitialized gallery object. The implementor should allocate this object during the function call. The user is responsible for freeing the object by calling *janice_free_gallery*.

Example

```
const size_t buffer_len = K; // Where K is the known length of the buffer
unsigned char buffer[buffer_len];

FILE* file = fopen("serialized.gallery", "r");
fread(buffer, 1, buffer_len, file);

JaniceGallery gallery = NULL; // best practice to initialize to NULL
janice_deserialize_gallery(buffer, buffer_len, gallery);

fclose(file);
```

9.4.10 janice_read_gallery

Read a gallery from a file on disk. This method is functionally equivalent to the following-

```
const size_t buffer_len = K; // Where K is the known length of the buffer
uint8_t buffer[buffer_len];

FILE* file = fopen("serialized.gallery", "r");
fread(buffer, 1, buffer_len, file);

JaniceGallery gallery = NULL; // best practice to initialize to NULL
janice_deserialize_gallery(buffer, buffer_len, gallery);

fclose(file);
```

It is provided for memory efficiency and ease of use when reading from disk.

Signature

```
JANICE_EXPORT JaniceError janice_read_gallery(const char* filename,
                                              JaniceGallery* gallery);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
filename	const char*	The path to a file on disk
gallery	*JaniceGallery**	An uninitialized gallery object. The implementor should allocate this object during the function call. The user is responsible for freeing this object by calling *janice_free_gallery*.

Example

```
JaniceGallery gallery = NULL;
if (janice_read_gallery("example.gallery", &gallery) != JANICE_SUCCESS)
    // ERROR!
```

9.4.11 janice_write_gallery

Write a gallery to a file on disk. This method is functionally equivalent to the following-

```
JaniceGallery gallery; // Where gallery is a valid gallery created previously.

uint8_t buffer = NULL;
size_t buffer_len;
janice_serialize_gallery(gallery, &buffer, &buffer_len);

FILE* file = fopen("serialized.gallery", "w+");
fwrite(buffer, 1, buffer_len, file);

fclose(file);
```

It is provided for memory efficiency and ease of use when writing to disk.

Signature

```
JANICE_EXPORT JaniceError janice_write_gallery(JaniceConstGallery gallery,
                                              const char* filename);
```

ThreadSafety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery*	The gallery object to write to disk.
filename	const char*	The path to a file on disk

Example

```
JaniceGallery gallery; // Where gallery is a valid gallery created previously
if (janice_write_gallery(gallery, "example.gallery") != JANICE_SUCCESS)
    // ERROR!
```

9.4.12 janice_free_gallery

Free any memory associated with a *JaniceGalleryType* object.

Signature

```
JANICE_EXPORT JaniceError janice_free_gallery(JaniceGallery* gallery);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
gallery	*JaniceGallery**	A gallery object to free.

Example

```
JaniceGallery gallery; // Where gallery is a valid gallery object created previously
if (janice_free_gallery(&gallery) != JANICE_SUCCESS)
    // ERROR!
```

9.4.13 janice_clear_template_ids

Free any memory associated with a of *JaniceTemplateIds* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_template_ids(JaniceTemplateIds* ids);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
ids	*JaniceTemplateIds**	A template ids objects to clear.

9.4.14 janice_clear_template_ids_group

Free any memory associated with a *JaniceTemplateIdsGroup* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_template_ids_group(JaniceTemplateIdsGroup*␣
↪group);
```

Parameters

Name	Type	Description
group	*JaniceTemplateIdsGroup**	A template ids group to clear.

CHAPTER

TEN

COMPARISON / SEARCH

10.1 Overview

This API defines two possible types of comparisons, 1:1 and 1:N. These are represented by the *janice_verify* and *janice_search* functions respectively. The API quantifies the relationship between two templates as a single number called a *Similarity Score*.

10.2 Structs

10.2.1 JaniceSimilarities

A structure representing a list of similarities.

Fields

Name	Type	Description
similarities	double*	An array of similarity objects.
length	size_t	The number of elements in `similarities`.

10.2.2 JaniceSimilaritiesGroup

A structure representing a list of *JaniceSimilarities* objects.

Fields

Name	Type	Description
group	*JaniceSimilarities**	An array of similarities objects.
length	size_t	The number of elements in `group`.

10.3 Functions

10.3.1 janice_verify

Compare two templates with the difference expressed as a similarity score.

Signature

```
JANICE_EXPORT JaniceError janice_verify(const JaniceConstTemplate reference,
                                        const JaniceConstTemplate verification,
                                        double* similarity);
```

Thread Safety

This function is *Reentrant*.

Similarity Score

This API expects that the comparison of two templates results in a single value that quantifies the similarity between them. A similarity score is constrained by the following requirements:

1. Higher scores indicate greater similarity
2. Scores can be asymmetric. This mean verify(a, b) does not necessarily equal verify(b, a)

Parameters

Name	Type	Description
reference	const *JaniceTemplate*	A reference template. This template was created with the `Janice1Reference` role.
verification	const *JaniceTemplate*	A verification template. This this template was created with the `Janice1Verification` role.
similarity	double*	A similarity score. See *Similarity Score*. Memory for this object should be managed by the user. The implementation should assume this points to a valid object that it can overwrite.

Example

```
JaniceTemplate reference; // Where reference is a valid template object created
                          // previously
JaniceTemplate verification; // Where verification is a valid template object
                             // created previously
JaniceSimilarity similarity;
if (janice_verify(reference, verification, &similarity) != JANICE_SUCCESS)
    // ERROR!
```

10.3.2 janice_verify_batch

Compute a batch of reference templates with a batch of verification templates. The ith in the reference batch is compared with the ith template in the verification batch. Batch processing can often be more efficient than serial processing, particularly if a GPU or co-processor is being utilized. This function reports per-comparison error codes. Depending on the batch policy given, it will return one of JANICE_SUCCESS if no errors occured, or JANICE_BATCH_ABORTED_EARLY or JANICE_BATCH_FINISHED_WITH_ERRORS if errors occured within the batch. In either case, any computation marked JANICE_SUCCESS in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_verify_batch(const JaniceTemplates* references,
                                              const JaniceTemplates* verifications,
                                              const JaniceContext* context,
                                              JaniceSimilarities* similarities,
                                              JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
references	const *JaniceTemplates**	An array of reference templates. Each template was created with the Janice1Reference role.
verifications	const *JaniceTemplates**	An array of verification templates. Each template was created with the Janice1Verification role. The number of elements in verifications must equal the number of elements in references.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
similarities	*JaniceSimilarities**	A struct to hold the output similarity scores. There must be the same number of similarity scores output as there are references and verifications. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_similarities*.
errors	*JaniceErrors**	A struct to hold per-comparison error codes. There must be the same number of errors as there are references and verifications unless the call aborted early, in which case there can be less. The ith error code should give the status of the ith comparison. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

10.3.3 janice_search

Compute 1-N search results between a query template object and a target gallery object. When running searches, users will often only want the top N results, or will only want results above a predefined threshold. This function must respect the `threshold` and `max_returns` fields of a *JaniceContext* object to facilitate these use cases. Implementors must always respect the passed threshold (i.e. a score below the given threshold should never be returned). If users would not like to specify a threshold they can set the member to `-DOUBLE_MAX`. If the `max_returns` member is non-zero implementors should respect both the threshold and the number of desired returns (i.e. return the top K scores above the given threshold). Users who would like to see all valid returns should set `max_returns` to 0.

This function allocates two structures with the same number of elements. `similarities` is a *JaniceSimilarities* object with an arra of *Similarity Score*, sorted in descending order. The second is a *JaniceTemplateIds* where the `ith` template id gives the unique identifier for the gallery template that produces the `ith` similarity score when compared with the probe.

Signature

```
JANICE_EXPORT JaniceError janice_search(const JaniceTemplate* probe,
                                        const JaniceGallery* gallery,
                                        const JaniceContext* context,
                                        JaniceSimilarities* similarities,
                                        JaniceTemplateIds* ids);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
probe	const *JaniceTemplate**	A query template. The template was created with the `Janice1NProbe` role.
gallery	const *JaniceGallery**	A gallery object to search against.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
similarities	*JaniceSimilarities**	A structure to hold the output similarity scores, sorted in descending order. This structure should have the same number of elements as `ids`. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_similarities*.
ids	*JaniceTemplateIds**	A structure to hold the gallery template ids associated with the `similarities`. This structure should have the same number of elements as `similarities`. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_template_ids*.

Example

```
JaniceTemplate probe;  // Where probe is a valid template object created
                       // previously
JaniceGallery gallery; // Where gallery is a valid gallery object created
                       // previously

JaniceContext context = nullptr;
if (janice_create_context(JaniceDetectAll, // detection policy, this shouldn't impact␣
↪search
                          0, // min_object_size, this shouldn't impact search
                          Janice1NProbe, // enrollment type, this shouldn't impact␣
↪search
                          0.7, // threshold, get all matches scoring above 0.7
                          50, // max_returns, get the top 50 matches scoring above␣
↪the set threshold
                          0, // hint, this shouldn't impact search
                          &context) != JANICE_SUCCESS)
    // ERROR!

JaniceSimilarities similarities;
JaniceTemplateIds ids;

// Run search
if (janice_search(probe, gallery, context, &similarities, &ids) != JANICE_SUCCESS)
    // ERROR!
```

10.3.4 janice_search_batch

Compute 1-N search results between a batch of probe templates and a single gallery. Given `N` probe templates in a batch, this function should return a single *JaniceSimilaritiesGroup* with N sublists and a single *JaniceTemplateIdsGroup* with N sublists. Each sublist must conform to the behavior defined in *janice_search*. Batch processing can often be more efficient than serial processing, particularly if a GPU or co-processor is being utilized. This function reports per-comparison error codes. Depending on the batch policy given, it will return one of `JANICE_SUCCESS` if no errors occured, or `JANICE_BATCH_ABORTED_EARLY` or `JANICE_BATCH_FINISHED_WITH_ERRORS` if errors occured within the batch. In either case, any computation marked `JANICE_SUCCESS` in the output should be considered valid output.

Signature

```
JANICE_EXPORT JaniceError janice_search_batch(const JaniceTemplates* probes,
                                              const JaniceGallery* gallery,
                                              const JaniceContext* context,
                                              JaniceSimilaritiesGroup* similarities,
                                              JaniceTemplateIdsGroup* ids,
                                              JaniceErrors* errors);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
probes	const *JaniceTemplates**	An array of probe templates to search with. Each template was created with the `Janice1NProbe` role.
gallery	const *JaniceGallery**	The gallery to search against.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
similarities	*JaniceSimilaritiesGroup**	A structure to hold the output similarities. Given `N` probes, there should be `N` sublists in the output, where the `ith` sublist gives the similarity scores of the `ith` probe. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_similarities_group*.
ids	*JaniceTemplateIdsGroup**	A structure to hold the output template ids. Given `N` probes, there should be :code‘N‘ sublists in the output, where the `ith` sublist gives the gallery template ids of the `ith` probe. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_template_ids_group*.
errors	*JaniceErrors**	A struct to hold per-search error codes. There must be the same number of errors as there are `probes` unless the call aborted early, in which case there can be less. The `ith` error code should give the status of the `ith` search. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is responsible for clearing the object by calling *janice_clear_errors*.

10.3.5 janice_clear_similarities

Free any memory associated with a *JaniceSimilarities* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_similarities(JaniceSimilarities* similarities);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
similarities	*JaniceSimilarities**	An similarities object to clear.

10.3.6 janice_clear_similarities_group

Free any memory associated with a *JaniceSimilaritiesGroup* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_similarities_group(JaniceSimilaritiesGroup*␣
↪group);
```

Parameters

Name	Type	Description
group	*JaniceSimilaritiesGroup**	A similarities group to clear.

CHAPTER

ELEVEN

CLUSTERING

11.1 Overview

This API defines clustering is the automatic and unsupervised combination of unlabelled templates into groups of like templates. What constitutes likeness is heavily dependent on the use case and context in question. One example when dealing with faces is grouping based on identity, where all faces belonging to a single individual are placed in a cluster.

11.2 Structs

11.2.1 JaniceClusterIds

A structure to represent a list of cluster ids objects.

Fields

Name	Type	Description
ids	uint64_t*	An array of cluster id objects.
length	size_t	The number of elements in `ids`

11.2.2 JaniceClusterIdsGroup

A structure to represent a list of *JaniceClusterIds* objects.

Fields

Name	Type	Description
group	*JaniceClusterIds**	An array of cluster ids objects.
length	size_t	The number of elements in `group`

11.2.3 JaniceClusterConfidences

A structure to represent a list of cluster confidence objects.

Fields

Name	Type	Description
confidences	double*	An array of cluster confidence objects.
length	size_t	The number of elements in `confidences`

11.2.4 JaniceClusterConfidencesGroup

A structure to represent a list of *JaniceClusterConfidences* objects.

Fields

Name	Type	Description
group	*JaniceClusterConfidences**	An array of cluster confidences objects.
length	size_t	The number of elements in `group`

11.3 Function

11.3.1 janice_cluster_media

Cluster a collection of media objects into groups. Each media object may contain 0 or more objects of interest. The output is arranged so that each output structure has `N` sublists where `N` is the number of input media and the `ith` sublist contains information for objects found in the `ith` media.

Cluster Confidence

Along with a cluster assignment, this API supports the concept of a cluster confidence. A cluster confidence is a value indicating a liklihood that the object of interest actually belongs to a cluster. For example, one possible implementation of a cluster confidence is the negative distance of an object from the cluster centroid. One use case for this value, is for end users to manually scrub cluster results by dynamically orphaning elements with lower confidence values. The cluster confidence is subject to the following contraints:

1. A higher value indicates greater confidence of cluster membership
2. No meaning can be assigned to an individual confidence, it is only relevant when being compared with other confidences generated by the same algorithm.

Signature

```
JANICE_EXPORT JaniceError janice_cluster_media(const JaniceMediaIterators* media,
                                               const JaniceContext* context,
                                               JaniceClusterIdsGroup* cluster_ids,
                                               JaniceClusterConfidencesGroup* cluster_
↪confidences,
                                               JaniceDetectionsGroup* detections);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
media	const *JaniceMediaIterators**	An array of media to cluster. After the function call, each iterator in the array will exist in an undefined state. A user should call *reset* on each iterator before reusing them.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
cluster_ids	*JaniceClusterIdsGroup**	An output structure to hold cluster ids. Objects with the same cluster id are members of the same cluster. This structure must have `N` sublists, where `N` is the number of elements in `media`. The `ith` sublist contains cluster ids for all objects of interest found in the `ith` media. If no objects of interest are found in a media then the corresponding sublist should have length 0. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_cluster_ids_group*.
cluster_confidences	*JaniceClusterConfidencesGroup**	An output structure to hold *Cluster Confidence*. This structure must have `N` sublists, where `N` is the number of elements in `media`. The `ith` sublist contains cluster confidences for all objects of interest found in the `ith` media. The `jth` confidence in the `ith` sublist refers to the same object as the `jth` id in the `ith` sublist of `ids`. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_cluster_confidences_group*.
detections	*JaniceDetectionsGroup**	Location information for each clustered object. This structure must have `N` sublists, where `N` is the number of elements in `media`. The `ith` sublist contains tracks for all objects of interest found in the `ith` media. The `jth` track in the `ith` sublist refers to the same object as the `jth` id in the `ith` sublist of `ids` and the `jth` confidence in the `ith` sublist of `cluster_confidences`. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_detections_group*.

11.3.2 janice_cluster_templates

Cluster a collection of template objects into groups.

Signature

```
JANICE_EXPORT JaniceError janice_cluster_templates(const JaniceTemplates* tmpls,
                                                   const JaniceContext* context,
                                                   JaniceClusterIds* cluster_ids,
                                                   JaniceClusterConfidences* cluster_
↪confidences);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
tmpls	const *JaniceTemplates**	An array of templates to cluster. Each template was created with the `JaniceCluster` role.
context	const *JaniceContext**	A context object with relevant hyperparameters set. Memory for the object should be managed by the user. The implementation should assume this points to a valid object.
cluster_ids	*JaniceClusterIds**	An output structure to hold cluster ids. Templates assigned the same cluster id are members of the same cluster. This structure must have the same number of elements as `tmpls`. The `ith` cluster id corresponds to the `ith` template object. Objects that can't be clustered should be assigned a unique cluster id. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_cluster_ids*.
cluster_confidences	*JaniceClusterConfidences**	An output structure to hold *Cluster Confidence*. This structure must have the same number of elements as `tmpls`. The `ith` cluster confidence corresponds to the `ith` template object. The user is responsible for allocating memory for the struct before the function call. The implementor is responsbile for allocating and filling internal members. The user is required to clear the struct by calling *janice_clear_cluster_confidences*.

11.3.3 janice_clear_cluster_ids

Free any memory associated with a of *JaniceClusterIds* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_cluster_ids(JaniceClusterIds* ids);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
ids	*JaniceClusterIds**	A cluster ids object to clear.

11.3.4 janice_clear_cluster_ids_group

Free any memory associated with a *JaniceClusterIdsGroup* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_cluster_ids_group(JaniceClusterIdsGroup*␣
↪group);
```

Parameters

Name	Type	Description
group	*JaniceClusterIdsGroup**	A cluster ids group to clear.

11.3.5 janice_clear_cluster_confidences

Free any memory associated with a of *JaniceClusterConfidences* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_cluster_confidences(JaniceClusterConfidences*␣
↪confidences);
```

Thread Safety

This function is *Reentrant*.

Parameters

Name	Type	Description
confidences	*JaniceClusterConfidences**	A cluster confidences object to clear.

11.3.6 janice_clear_cluster_confidences_group

Free any memory associated with a *JaniceClusterConfidencesGroup* object.

Signature

```
JANICE_EXPORT JaniceError janice_clear_cluster_confidences_
↪group(JaniceClusterConfidencesGroup* group);
```

Parameters

Name	Type	Description
group	*JaniceClusterConfidencesGroup**	A cluster confidences group to clear.

CHAPTER

TWELVE

LICENSE

```
/*******************************************************************************
 * Copyright (c) 2013 Noblis, Inc.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a
 * copy of this software and/or associated documentation files (the
 * "Materials"), to deal in the Materials without restriction, including
 * without limitation the rights to use, copy, modify, merge, publish,
 * distribute, sublicense, and/or sell copies of the Materials, and to
 * permit persons to whom the Materials are furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included
 * in all copies or substantial portions of the Materials.
 *
 * THE MATERIALS ARE PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
 * EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
 * IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
 * CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
 * TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
 * MATERIALS OR THE USE OR OTHER DEALINGS IN THE MATERIALS.
 ******************************************************************************/
```

The JanICE API is a *C* API that provides a common interface between computer vision algorithms and agencies and entities that would like to use them. The API consists of a core header file defining required *C* functions. It also defines a number of interfaces to other languages on top of the *C* API.

CHAPTER THIRTEEN

ABOUT

Computer vision is a rapidly expanding and improving field that has seen significant progress in it's capabilities over the past decade. Government agencies can leverage computer vision algorithms to better understand images and videos that they ingest. This in turn can lead to improved response times, increased public safety, and numerous other benefits. The JanICE API provides a common framework that commercial vendors and government agencies can use to ease integration between algorithms and use cases. The API aims to cover a number of different Computer Vision subproblems. At this time, these problems include:

- Face Recognition

Some function calls serve multiple use cases in different ways. In those cases the function documentation strives to clearly indicate the differences. If no differences are indicated it means that the function is universal in that it applies the same to each subproblem addressed by the API.

This work is being sponsored by The Department of Homeland Security; Science and Technology Directorate.

CHAPTER

FOURTEEN

FOCUS AREAS

14.1 Face Recognition

Facial recognition has emerged as a key technology for government agencies to efficiently triage and analyze large data streams. A large ecosystem of facial recognition algorithms already exists from a variety of sources including commercial vendors, government programs and academia. However, integrating this important technology into existing technology stacks is a difficult and expensive endeavor. The JanICE API aims to address this problem by functioning as a compatibility layer between users and the algorithms. Users can write their applications on “top” of the API while algorithm providers will implement their algorithms “beneath” the API. This means that users can write their applications independent of any single FR algorithm and gives them the freedom to select the algorithm or algorithms that best serve their specific use case without worrying about integration. Algorithm providers will be able to serve their algorithms across teams and agencies without having to integrate with the different tools and services of each specific team.

CHAPTER FIFTEEN

LICENSE

The API is provided under the MIT *License* and is *free for academic and commercial use*.

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search