

PBImgAnalysis

P. Baillehache

April 29, 2019

Contents

1	Interface	2
2	Code	15
2.1	pbimganalysis.c	15
2.2	pbimganalysis-inline.c	65
3	Makefile	76
4	Unit tests	76
5	Unit tests output	86
5.1	K-Means clustering on RGBA space	87
5.2	ImgSegmentor	93
5.2.1	Test 01	93
5.2.2	Test 02	94
5.2.3	Test 03	95
5.2.4	Test 04	96

Introduction

PBImgAnalysis is a C library providing structures and functions to perform various data analysis on images.

It implements the following algorithms:

- K-means clustering on the RGBA space of pixels in a user defined radius
- Intersection over Union (aka Jaccard index)

- ImgSegmentor, a multiclass multimodal image segmentation algorithm based on heuristics and NeuraNet

It uses the PBErr, PBDataAnalysis, GenBrush, GDataSet, GenAlg, NeuraNet, ResPublish libraries.

1 Interface

```
// ===== PBIMGANALYSIS.H =====

#ifndef PBIMGANALYSIS_H
#define PBIMGANALYSIS_H

// ===== Include =====

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <execinfo.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <signal.h>
#include "pberr.h"
#include "genbrush.h"
#include "pbdataanalysis.h"
#include "genalg.h"
#include "neuranet.h"
#include "gdataset.h"
#include "respublish.h"

// ----- ImgKMeansClusters -----

// ===== Define =====

// ===== Data structure =====

typedef struct ImgKMeansClusters {
    // Image on which the clustering is applied
    // Uses the GBSurfaceFinalPixels
    const GenBrush* _img;
    // Clusters result of the search
    KMeansClusters _kmeansClusters;
    // Size of the considered cell in the image around a given position
    // is equal to (_size * 2 + 1)
    int _size;
} ImgKMeansClusters;

// ===== Functions declaration =====

// Create a new ImgKMeansClusters for the image 'img' and with seed 'seed'
// and type 'type' and a cell size equal to 2*'size'+1
ImgKMeansClusters ImgKMeansClustersCreateStatic(
    const GenBrush* const img, const KMeansClustersSeed seed,
    const int size);
```

```

// Free the memory used by a ImgKMeansClusters
void ImgKMeansClustersFreeStatic(ImgKMeansClusters* const that);

// Get the GenBrush of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const GenBrush* IKMCImg(const ImgKMeansClusters* const that);

// Set the GenBrush of the ImgKMeansClusters 'that' to 'img'
#if BUILDMODE != 0
inline
#endif
void IKMCSetImg(ImgKMeansClusters* const that, const GenBrush* const img);

// Set the size of the cells of the ImgKMeansClusters 'that' to
// 2*'size'+1
#if BUILDMODE != 0
inline
#endif
void IKMCSetSizeCell(ImgKMeansClusters* const that, const int size);

// Get the number of cluster of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int IKMCGetK(const ImgKMeansClusters* const that);

// Get the size of the cells of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int IKMCGetSizeCell(const ImgKMeansClusters* const that);

// Get the KMeansClusters of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const KMeansClusters* IKMCKMeansClusters(
    const ImgKMeansClusters* const that);

// Search for the 'K' clusters in the image of the
// ImgKMeansClusters 'that'
void IKMCSearch(ImgKMeansClusters* const that, const int K);

// Print the ImgKMeansClusters 'that' on the stream 'stream'
void IKMCPrintln(const ImgKMeansClusters* const that,
    FILE* const stream);

// Get the index of the cluster at position 'pos' for the
// ImgKMeansClusters 'that'
int IKMCGetId(const ImgKMeansClusters* const that,
    const VecShort2D* const pos);

// Get the GBPixel equivalent to the cluster at position 'pos'
// for the ImgKMeansClusters 'that'
GBPixel IKMCGetPixel(const ImgKMeansClusters* const that,
    const VecShort2D* const pos);

// Convert the image of the ImageKMeansClusters 'that' to its clustered
// version
// IKMCSearch must have been called previously

```

```

void IKMCcluster(const ImgKMeansClusters* const that);

// Load the IKMC 'that' from the stream 'stream'
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCLoad(ImgKMeansClusters* that, FILE* const stream);

// Save the IKMC 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCSave(const ImgKMeansClusters* const that,
    FILE* const stream, const bool compact);

// Function which return the JSON encoding of 'that'
JSONNode* IKMCEncodeAsJSON(const ImgKMeansClusters* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool IKMCDecodeAsJSON(ImgKMeansClusters* that,
    const JSONNode* const json);

// ===== Polymorphism =====

// ----- General functions -----

// Return the Jaccard index (aka intersection over union) of the
// images 'that' and 'tho' for pixels of color 'rgba'
// 'that' and 'tho' must have same dimensions
float IntersectionOverUnion(const GenBrush* const that,
    const GenBrush* const tho, const GBPixel* const rgba);

// Return the similarity coefficient of the images 'that' and 'tho'
// (i.e. the sum of the distances of pixels at the same position
// over the whole image)
// Return a value in [0.0, 1.0], 1.0 means the two images are
// identical, 0.0 means they are binary black and white with each
// pixel in one image the opposite of the corresponding pixel in the
// other image.
// 'that' and 'tho' must have same dimensions
float GBSimilarityCoeff(const GenBrush* const that,
    const GenBrush* const tho);

// ----- ImgSegmentor -----

// ===== Define =====

#define IS_TRAINTXTOMETER_LINE1 "Epoch xxxxx/xxxxx Entity xxx/xxx\n"
#define IS_TRAINTXTOMETER_FORMAT1 "Epoch %05ld/%05ld Entity %03d/%03d\n"
#define IS_EVALTXTOMETER_LINE1 "Sample xxxxx/xxxxx\n"
#define IS_EVALTXTOMETER_FORMAT1 "Sample %05ld/%05ld\n"

#define IS_CHECKPOINTFILENAME "checkpoint.json"

// ===== Data structure =====

typedef struct ImgSegmentor {
    // Tree of criterion
    GenTree _criteria;
    // Number of segmentation class
    int _nbClass;
    // Flag to apply or not the binarization on result of prediction

```

```

// false by default
bool _flagBinaryResult;
// Threshold value for the binarization of result of prediction
// If the result of prediction is above the threshold then
// the result is considered equal to 1.0 else it is considered equal
// to -1.0
// 0.5 by default
float _thresholdBinaryResult;
// Nb of epoch for training, 1 by default
unsigned int _nbEpoch;
// Size pool for training
// By default GENALG_NBENTITIES
int _sizePool;
// Nb min of adns
int _sizeMinPool;
// Nb max of adns
int _sizeMaxPool;
// Nb elite for training
// By default GENALG_NBELITES
int _nbElite;
// Threshold to stop the training once
float _targetBestValue;
// Flag to memorize if we display info during training with a TextOMeter
bool _flagTextOMeter;
// TextOMeter to display info during training
TextOMeter* _textOMeter;
// Strings for the TextOMeter
char _line1[50];
char _line2[50];
} ImgSegmentor;

typedef struct ImgSegmentorPerf {
    // Accuracy
    float _accuracy;
} ImgSegmentorPerf;

typedef struct ImgSegmentorTrainParam {
    // Nb of epochs
    int _nbEpoch;
} ImgSegmentorParam;

typedef enum ISCType {
    ISCType_RGB, ISCType_RGB2HSV, ISCType_Dust, ISCType_Tex
} ISCType;

typedef struct ImgSegmentorCriterion {
    // Type of criterion
    ISCType _type;
    // Nb of class
    int _nbClass;
} ImgSegmentorCriterion;

typedef struct ImgSegmentorCriterionRGB {
    // ImgSegmentorCriterion
    ImgSegmentorCriterion _criterion;
    // NeuraNet model
    NeuraNet* _nn;
} ImgSegmentorCriterionRGB;

typedef struct ImgSegmentorCriterionRGB2HSV {
    // ImgSegmentorCriterion
    ImgSegmentorCriterion _criterion;

```

```

} ImgSegmentorCriterionRGB2HSV;

typedef struct ImgSegmentorCriterionDust {
    // ImgSegmentorCriterion
    ImgSegmentorCriterion _criterion;
    // Dust size for each class
    VecLong* _size;
} ImgSegmentorCriterionDust;

typedef struct ImgSegmentorCriterionTex {
    // ImgSegmentorCriterion
    ImgSegmentorCriterion _criterion;
    // NeuraNet model
    NeuraNet* _nn;
    // Rank (nb of hidden layers in the NeuraNet)
    int _rank;
    // Size (consider from 3^size x 3^size to 1x1 square pixels fragments)
    int _size;
} ImgSegmentorCriterionTex;

// ===== Functions declaration =====

// Create a new static ImgSegmentor with 'nbClass' output
ImgSegmentor ImgSegmentorCreateStatic(int nbClass);

// Create a new ImgSegmentor with 'nbClass' output
ImgSegmentor* ImgSegmentorCreate(int nbClass);

// Free the memory used by the static ImgSegmentor 'that'
void ImgSegmentorFreeStatic(ImgSegmentor* that);

// Free the memory used by the ImgSegmentor 'that'
void ImgSegmentorFree(ImgSegmentor** that);

// Return the nb of criterion of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
long ISGetNbCriterion(const ImgSegmentor* const that);

// Set the flag memorizing if the TextOMeter is displayed for
// the ImgSegmentor 'that' to 'flag'
void ISSetFlagTextOMeter(ImgSegmentor* const that, bool flag);

// Return the flag for the TextOMeter of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
bool ISGetFlagTextOMeter(const ImgSegmentor* const that);

// Refresh the content of the TextOMeter attached to the
// ImgSegmentor 'that'
void ISUpdateTextOMeter(const ImgSegmentor* const that);

// Add a new ImageSegmentorCriterionRGB to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
// Return the added criterion if successful, null else
#if BUILDMODE != 0
inline
#endif
ImgSegmentorCriterionRGB* ISAddCriterionRGB(ImgSegmentor* const that,

```

```

    void* const parent);

// Add a new ImageSegmentorCriterionTex to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
// Return the added criterion if successful, null else
#if BUILDMODE != 0
inline
#endif
ImgSegmentorCriterionTex* ISAddCriterionTex(ImgSegmentor* const that,
    void* const parent, const int rank, const int size);

// Add a new ImageSegmentorCriterionRGB2HSV to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
// Return the added criterion if successful, null else
#if BUILDMODE != 0
inline
#endif
ImgSegmentorCriterionRGB2HSV* ISAddCriterionRGB2HSV(
    ImgSegmentor* const that, void* const parent);

// Return the nb of classes of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetNbClass(const ImgSegmentor* const that);

// Return the flag controlling the binarization of the result of
// prediction of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
bool ISGetFlagBinaryResult(const ImgSegmentor* const that);

// Return the threshold controlling the binarization of the result of
// prediction of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
float ISGetThresholdBinaryResult(const ImgSegmentor* const that);

// Return the threshold controlling the stop of the training
#if BUILDMODE != 0
inline
#endif
float ISGetTargetBestValue(const ImgSegmentor* const that);

// Set the threshold controlling the stop of the training to 'val'
// Clip the value to [0.0, 1.0]
#if BUILDMODE != 0
inline
#endif
void ISSetTargetBestValue(ImgSegmentor* const that, const float val);

// Set the flag controlling the binarization of the result of
// prediction of the ImgSegmentor 'that' to 'flag'
#if BUILDMODE != 0
inline
#endif
void ISSetFlagBinaryResult(ImgSegmentor* const that,
    const bool flag);

```

```

// Return the number of epoch for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
unsigned int ISGetNbEpoch(const ImgSegmentor* const that);

// Set the number of epoch for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetNbEpoch(ImgSegmentor* const that, unsigned int nb);

// Return the size of the pool for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizePool(const ImgSegmentor* const that);

// Set the size of the pool for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizePool(ImgSegmentor* const that, int nb);

// Return the nb of elites for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetNbElite(const ImgSegmentor* const that);

// Set the nb of elites for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetNbElite(ImgSegmentor* const that, int nb);

// Return the max nb of adns of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizeMaxPool(const ImgSegmentor* const that);

// Return the min nb of adns of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizeMinPool(const ImgSegmentor* const that);

// Set the min nb of adns of the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizeMaxPool(ImgSegmentor* const that, const int nb);

// Set the min nb of adns of the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizeMinPool(ImgSegmentor* const that, const int nb);

// Set the threshold controlling the binarization of the result of

```



```

// prediction of the ImgSegmentor 'that' to 'threshold'
#if BUILDMODE != 0
inline
#endif
void ISSetThresholdBinaryResult(ImgSegmentor* const that,
    const float threshold);

// Make a prediction on the GenBrush 'img' with the ImgSegmentor 'that'
// Return an array of pointer to GenBrush, one per output class, in
// greyscale, where the color of each pixel indicates the detection of
// the corresponding class at the given pixel, white equals no
// detection, black equals detection, 50% grey equals "don't know"
GenBrush** ISPredict(const ImgSegmentor* const that,
    const GenBrush* const img);

// Return the nb of criterion of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
const GenTree* ISCriteria(const ImgSegmentor* const that);

// Train the ImageSegmentor 'that' on the data set 'dataSet' using
// the data of the first category in 'dataSet'. If the data set has a
// second category it will be used for validation
// random must have been called before calling ISTrain
void ISTrain(ImgSegmentor* const that,
    const GDataSetGenBrushPair* const dataset);

// Evaluate the ImageSegmentor 'that' on the data set 'dataSet' using
// the data of the 'iCat' category in 'dataSet'
// random must have been called before calling ISTrain
// Return a value in [0.0, 1.0], 0.0 being worst and 1.0 being best
float ISEvaluate(ImgSegmentor* const that,
    const GDataSetGenBrushPair* const dataset, const int iCat);

// Load the ImgSegmentor from the stream
// If the ImgSegmentor is already allocated, it is freed before loading
// Return true upon success else false
bool ISLoad(ImgSegmentor* that, FILE* const stream);

// Save the ImgSegmentor to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success else false
bool ISSave(const ImgSegmentor* const that,
    FILE* const stream, const bool compact);

// Function which return the JSON encoding of 'that'
JSONNode* ImgSegmentorEncodeAsJSON(const ImgSegmentor* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool ImgSegmentorDecodeAsJSON(ImgSegmentor* that,
    const JSONNode* const json);

// Create a new static ImgSegmentorCriterion with 'nbClass' output
// and the type of criterion 'type'
ImgSegmentorCriterion ImgSegmentorCriterionCreateStatic(int nbClass,
    ISCType type);

// Free the memory used by the static ImgSegmentorCriterion 'that'
void ImgSegmentorCriterionFreeStatic(ImgSegmentorCriterion* that);

```

```

// Make the prediction on the 'input' values by calling the appropriate
// function according to the type of criterion
// 'input' 's format is width*height*3, values in [0.0, 1.0]
// Return values are width*height*nbClass, values in [-1.0, 1.0]
VecFloat* ISCPredict(const ImgSegmentorCriterion* const that,
    const VecFloat* input, const VecShort2D* const dim);

// Return the nb of class of the ImgSegmentorCriterion 'that'
#ifdef BUILDMODE != 0
inline
#endif
int _ISCGetNbClass(const ImgSegmentorCriterion* const that);

// Return the number of int parameters for the criterion 'that'
long _ISCGetNbParamInt(const ImgSegmentorCriterion* const that);

// Return the number of float parameters for the criterion 'that'
long _ISCGetNbParamFloat(const ImgSegmentorCriterion* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void _ISCSetBoundsAdnInt(const ImgSegmentorCriterion* const that,
    GenAlg* const ga, const long shift);

// Set the bounds of float parameters for training of the criterion 'that'
void _ISCSetBoundsAdnFloat(const ImgSegmentorCriterion* const that,
    GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void _ISCSetAdnInt(const ImgSegmentorCriterion* const that,
    const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void _ISCSetAdnFloat(const ImgSegmentorCriterion* const that,
    const GenAlgAdn* const adn, const long shift);

// ---- ImgSegmentorCriterionRGB

// Create a new ImgSegmentorCriterionRGB with 'nbClass' output
ImgSegmentorCriterionRGB* ImgSegmentorCriterionRGBCreate(
    const int nbClass);

// Free the memory used by the ImgSegmentorCriterionRGB 'that'
void ImgSegmentorCriterionRGBFree(ImgSegmentorCriterionRGB** that);

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionRGB that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCRGBPredict(const ImgSegmentorCriterionRGB* const that,
    const VecFloat* input, const VecShort2D* const dim);

// Return the number of int parameters for the criterion 'that'
long ISCRGBGetNbParamInt(const ImgSegmentorCriterionRGB* const that);

// Return the number of float parameters for the criterion 'that'
long ISCRGBGetNbParamFloat(const ImgSegmentorCriterionRGB* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void ISCRGBSetBoundsAdnInt(const ImgSegmentorCriterionRGB* const that,
    GenAlg* const ga, const long shift);

// Set the bounds of float parameters for training of the criterion 'that'

```

```

void ISCRGBSetBoundsAdnFloat(const ImgSegmentorCriterionRGB* const that,
    GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void ISCRGBSetAdnInt(const ImgSegmentorCriterionRGB* const that,
    const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void ISCRGBSetAdnFloat(const ImgSegmentorCriterionRGB* const that,
    const GenAlgAdn* const adn, const long shift);

// Return the NeuraNet of the ImgSegmentorCriterionRGB 'that'
#ifdef BUILDMODE != 0
inline
#endif
const NeuraNet* ISCRGBNeuraNet(
    const ImgSegmentorCriterionRGB* const that);

// ---- ImgSegmentorCriterionRGB2HSV

// Create a new ImgSegmentorCriterionRGB2HSV with 'nbClass' output
ImgSegmentorCriterionRGB2HSV* ImgSegmentorCriterionRGB2HSVCreate(
    const int nbClass);

// Free the memory used by the ImgSegmentorCriterionRGB2HSV 'that'
void ImgSegmentorCriterionRGB2HSVFree(
    ImgSegmentorCriterionRGB2HSV** that);

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionRGB2HSV that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCRGB2HSVPredict(
    const ImgSegmentorCriterionRGB2HSV* const that,
    const VecFloat* input, const VecShort2D* const dim);

// Return the number of int parameters for the criterion 'that'
long ISCRGB2HSVGetNbParamInt(
    const ImgSegmentorCriterionRGB2HSV* const that);

// Return the number of float parameters for the criterion 'that'
long ISCRGB2HSVGetNbParamFloat(
    const ImgSegmentorCriterionRGB2HSV* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void ISCRGB2HSVSetBoundsAdnInt(
    const ImgSegmentorCriterionRGB2HSV* const that,
    GenAlg* const ga, const long shift);

// Set the bounds of float parameters for training of the criterion 'that'
void ISCRGB2HSVSetBoundsAdnFloat(
    const ImgSegmentorCriterionRGB2HSV* const that,
    GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void ISCRGB2HSVSetAdnInt(const ImgSegmentorCriterionRGB2HSV* const that,
    const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void ISCRGB2HSVSetAdnFloat(const ImgSegmentorCriterionRGB2HSV* const that,
    const GenAlgAdn* const adn, const long shift);

```

```

// ---- ImgSegmentorCriterionDust

// Create a new ImgSegmentorCriterionDust with 'nbClass' output
ImgSegmentorCriterionDust* ImgSegmentorCriterionDustCreate(
    const int nbClass);

// Free the memory used by the ImgSegmentorCriterionDust 'that'
void ImgSegmentorCriterionDustFree(
    ImgSegmentorCriterionDust** that);

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionDust that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCDustPredict(
    const ImgSegmentorCriterionDust* const that,
    const VecFloat* input, const VecShort2D* const dim);

// Return the number of int parameters for the criterion 'that'
long ISCDustGetNbParamInt(
    const ImgSegmentorCriterionDust* const that);

// Return the number of float parameters for the criterion 'that'
long ISCDustGetNbParamFloat(
    const ImgSegmentorCriterionDust* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void ISCDustSetBoundsAdnInt(
    const ImgSegmentorCriterionDust* const that,
    GenAlg* const ga, const long shift);

// Set the bounds of float parameters for training of the criterion 'that'
void ISCDustSetBoundsAdnFloat(
    const ImgSegmentorCriterionDust* const that,
    GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void ISCDustSetAdnInt(const ImgSegmentorCriterionDust* const that,
    const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void ISCDustSetAdnFloat(const ImgSegmentorCriterionDust* const that,
    const GenAlgAdn* const adn, const long shift);

// Return the dust size of the ImgSegmentorCriterionDust 'that' for
// the class 'iClass'
#ifdef BUILDMODE != 0
inline
#endif
long ISCDustSize(
    const ImgSegmentorCriterionDust* const that, const int iClass);

// Set the dust size of the ImgSegmentorCriterionDust 'that' for
// the class 'iClass' to 'size'
#ifdef BUILDMODE != 0
inline
#endif
void ISCDustSetSize(
    const ImgSegmentorCriterionDust* const that, const int iClass,
    const long size);

// ---- ImgSegmentorCriterionTex

```

```

// Create a new ImgSegmentorCriterionTex with 'nbClass' output,
// 'rank' hidden layers and 3^'size' x 3^'size' down to 1x1 square
// fragments of the image as input
ImgSegmentorCriterionTex* ImgSegmentorCriterionTexCreate(
    const int nbClass, const int rank, const int size);

// Free the memory used by the ImgSegmentorCriterionTex 'that'
void ImgSegmentorCriterionTexFree(ImgSegmentorCriterionTex** that);

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionTex that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCTexPredict(const ImgSegmentorCriterionTex* const that,
    const VecFloat* input, const VecShort2D* const dim);

// Return the number of int parameters for the criterion 'that'
long ISCTexGetNbParamInt(const ImgSegmentorCriterionTex* const that);

// Return the number of float parameters for the criterion 'that'
long ISCTexGetNbParamFloat(const ImgSegmentorCriterionTex* const that);

// Set the bounds of int parameters for training of the criterion 'that'
void ISCTexSetBoundsAdnInt(const ImgSegmentorCriterionTex* const that,
    GenAlg* const ga, const long shift);

// Set the bounds of float parameters for training of the criterion 'that'
void ISCTexSetBoundsAdnFloat(const ImgSegmentorCriterionTex* const that,
    GenAlg* const ga, const long shift);

// Set the values of int parameters for training of the criterion 'that'
void ISCTexSetAdnInt(const ImgSegmentorCriterionTex* const that,
    const GenAlgAdn* const adn, const long shift);

// Set the values of float parameters for training of the criterion 'that'
void ISCTexSetAdnFloat(const ImgSegmentorCriterionTex* const that,
    const GenAlgAdn* const adn, const long shift);

// Return the NeuraNet of the ImgSegmentorCriterionTex 'that'
#ifdef BUILDMODE != 0
inline
#endif
const NeuraNet* ISCTexNeuraNet(
    const ImgSegmentorCriterionTex* const that);

// Return the rank of the ImgSegmentorCriterionTex 'that'
#ifdef BUILDMODE != 0
inline
#endif
int ISCTexGetRank(const ImgSegmentorCriterionTex* const that);

// Return the size of the ImgSegmentorCriterionTex 'that'
#ifdef BUILDMODE != 0
inline
#endif
int ISCTexGetSize(const ImgSegmentorCriterionTex* const that);

// ===== Polymorphism =====

#define ISCGetNbClass(That) _Generic(That, \
    ImgSegmentorCriterion*: _ISCGetNbClass, \

```

```

const ImgSegmentorCriterion*: _ISCGetNbClass, \
ImgSegmentorCriterionRGB*: _ISCGetNbClass, \
const ImgSegmentorCriterionRGB*: _ISCGetNbClass, \
ImgSegmentorCriterionTex*: _ISCGetNbClass, \
const ImgSegmentorCriterionTex*: _ISCGetNbClass, \
ImgSegmentorCriterionRGB2HSV*: _ISCGetNbClass, \
const ImgSegmentorCriterionRGB2HSV*: _ISCGetNbClass, \
ImgSegmentorCriterionDust*: _ISCGetNbClass, \
const ImgSegmentorCriterionDust*: _ISCGetNbClass, \
default: PBErrInvalidPolymorphism) ((const ImgSegmentorCriterion*)That)

#define ISCGetNbParamInt(That) _Generic(That, \
    ImgSegmentorCriterion*: _ISCGetNbParamInt, \
    const ImgSegmentorCriterion*: _ISCGetNbParamInt, \
    ImgSegmentorCriterionRGB*: ISCRGBGetNbParamInt, \
    const ImgSegmentorCriterionRGB*: ISCRGBGetNbParamInt, \
    ImgSegmentorCriterionTex*: ISCTexGetNbParamInt, \
    const ImgSegmentorCriterionTex*: ISCTexGetNbParamInt, \
    ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVGetNbParamInt, \
    const ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVGetNbParamInt, \
    ImgSegmentorCriterionDust*: ISCDustGetNbParamInt, \
    const ImgSegmentorCriterionDust*: ISCDustGetNbParamInt, \
    default: PBErrInvalidPolymorphism) ((const ImgSegmentorCriterion*)That)

#define ISCGetNbParamFloat(That) _Generic(That, \
    ImgSegmentorCriterion*: _ISCGetNbParamFloat, \
    const ImgSegmentorCriterion*: _ISCGetNbParamFloat, \
    ImgSegmentorCriterionRGB*: ISCRGBGetNbParamFloat, \
    const ImgSegmentorCriterionRGB*: ISCRGBGetNbParamFloat, \
    ImgSegmentorCriterionTex*: ISCTexGetNbParamFloat, \
    const ImgSegmentorCriterionTex*: ISCTexGetNbParamFloat, \
    ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVGetNbParamFloat, \
    const ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVGetNbParamFloat, \
    ImgSegmentorCriterionDust*: ISCDustGetNbParamFloat, \
    const ImgSegmentorCriterionDust*: ISCDustGetNbParamFloat, \
    default: PBErrInvalidPolymorphism) ((const ImgSegmentorCriterion*)That)

#define ISCSetBoundsAdnInt(That, GenAlg, Shift) _Generic(That, \
    ImgSegmentorCriterion*: _ISCSetBoundsAdnInt, \
    const ImgSegmentorCriterion*: _ISCSetBoundsAdnInt, \
    ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnInt, \
    const ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnInt, \
    ImgSegmentorCriterionTex*: ISCTexSetBoundsAdnInt, \
    const ImgSegmentorCriterionTex*: ISCTexSetBoundsAdnInt, \
    ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetBoundsAdnInt, \
    const ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetBoundsAdnInt, \
    ImgSegmentorCriterionDust*: ISCDustSetBoundsAdnInt, \
    const ImgSegmentorCriterionDust*: ISCDustSetBoundsAdnInt, \
    default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, GenAlg, Shift)

#define ISCSetBoundsAdnFloat(That, GenAlg, Shift) _Generic(That, \
    ImgSegmentorCriterion*: _ISCSetBoundsAdnFloat, \
    const ImgSegmentorCriterion*: _ISCSetBoundsAdnFloat, \
    ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnFloat, \
    const ImgSegmentorCriterionRGB*: ISCRGBSetBoundsAdnFloat, \
    ImgSegmentorCriterionTex*: ISCTexSetBoundsAdnFloat, \
    const ImgSegmentorCriterionTex*: ISCTexSetBoundsAdnFloat, \
    ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetBoundsAdnFloat, \
    const ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetBoundsAdnFloat, \
    ImgSegmentorCriterionDust*: ISCDustSetBoundsAdnFloat, \
    const ImgSegmentorCriterionDust*: ISCDustSetBoundsAdnFloat, \

```

```

default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, GenAlg, Shift)

#define ISCSetsAdnInt(That, Adn, Shift) _Generic(That, \
    ImgSegmentorCriterion*: _ISCSetsAdnInt, \
    const ImgSegmentorCriterion*: _ISCSetsAdnInt, \
    ImgSegmentorCriterionRGB*: ISCRGBSetsAdnInt, \
    const ImgSegmentorCriterionRGB*: ISCRGBSetsAdnInt, \
    ImgSegmentorCriterionTex*: ISCTexSetsAdnInt, \
    const ImgSegmentorCriterionTex*: ISCTexSetsAdnInt, \
    ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetAdnInt, \
    const ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetAdnInt, \
    ImgSegmentorCriterionDust*: ISCDustSetAdnInt, \
    const ImgSegmentorCriterionDust*: ISCDustSetAdnInt, \
    default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, Adn, Shift)

#define ISCSetsAdnFloat(That, Adn, Shift) _Generic(That, \
    ImgSegmentorCriterion*: _ISCSetsAdnFloat, \
    const ImgSegmentorCriterion*: _ISCSetsAdnFloat, \
    ImgSegmentorCriterionRGB*: ISCRGBSetsAdnFloat, \
    const ImgSegmentorCriterionRGB*: ISCRGBSetsAdnFloat, \
    ImgSegmentorCriterionTex*: ISCTexSetsAdnFloat, \
    const ImgSegmentorCriterionTex*: ISCTexSetsAdnFloat, \
    ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetAdnFloat, \
    const ImgSegmentorCriterionRGB2HSV*: ISCRGB2HSVSetAdnFloat, \
    ImgSegmentorCriterionDust*: ISCDustSetAdnFloat, \
    const ImgSegmentorCriterionDust*: ISCDustSetAdnFloat, \
    default: PBErrInvalidPolymorphism) ( \
    (const ImgSegmentorCriterion*)That, Adn, Shift)

// ===== Inliner =====

#if BUILDMODE != 0
#include "pbimganalysis-inline.c"
#endif

#endif

```

2 Code

2.1 pbimganalysis.c

```

// ===== PBIMGANALYSIS.C =====

// ===== Include =====

#include "pbimganalysis.h"
#if BUILDMODE == 0
#include "pbimganalysis-inline.c"
#endif

// ----- ImgKMeansClusters -----

// ===== Define =====

// ===== Global variable =====

```

```

// Variable to handle the signal Ctrl-C during training
static volatile bool PBIA_CtrlC = false;

// ===== Functions declaration =====

// Get the input values for the pixel at position 'pos' according to
// the cell size of the ImgKMeansClusters 'that'
// The return is a VecFloat made of the sizeCell^2 pixels' value
// around pos ordered by ((r*256+g)*256+b)*256+a)
VecFloat* IKMCGetInputOverCell(const ImgKMeansClusters* const that,
    const VecShort2D* const pos);

// ===== Functions implementation =====

// Create a new ImgKMeansClusters for the image 'img' and with seed 'seed'
// and type 'type' and a cell size equal to 2*'size'+1
ImgKMeansClusters ImgKMeansClustersCreateStatic(
    const GenBrush* const img, const KMeansClustersSeed seed,
    const int size) {
#ifdef BUILDMODE == 0
    if (img == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'img' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (size < 0) {
        PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBImpAnalysisErr->_msg, "'size' is invalid (%d>=0)", size);
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // Declare the new ImgKMeansClusters
    ImgKMeansClusters that;
    // Set properties
    that._img = img;
    that._kmeansClusters = KMeansClustersCreateStatic(seed);
    that._size = size;
    // Return the new ImgKMeansClusters
    return that;
}

// Free the memory used by a ImgKMeansClusters
void ImgKMeansClustersFreeStatic(ImgKMeansClusters* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // Reset the GenBrush associated to the IKMC
    that->_img = NULL;
    // Free the memory used by the KMeansClusters
    KMeansClustersFreeStatic((KMeansClusters*)IKMCKMeansClusters(that));
}

// Search for the 'K' clusters in the image of the
// ImgKMeansClusters 'that'
void IKMCSearch(ImgKMeansClusters* const that, const int K) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;

```



```

        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
    if (K < 1) {
        PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBImgAnalysisErr->_msg, "'K' is invalid (%d>0)", K);
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    // Create a set to memorize the input over cells
    GSetVecFloat inputOverCells = GSetVecFloatCreateStatic();
    // Get the dimension of the image
    VecShort2D dim = GBGetDim(IKMCImg(that));
    // Loop on pixels
    VecShort2D pos = VecShortCreateStatic2D();
    do {
        // Get the KMeansClusters input over the cell
        VecFloat* inputOverCell = IKMCGetInputOverCell(that, &pos);
        // Add it to the inputs for the search
        GSetAppend(&inputOverCells, inputOverCell);
    } while (VecStep(&pos, &dim));
    // Search the clusters
    KMeansClustersSearch((KMeansClusters*)IKMCKMeansClusters(that),
        &inputOverCells, K);
    // Free the memory used by the input
    while (GSetNbElem(&inputOverCells) > 0) {
        VecFloat* v = GSetPop(&inputOverCells);
        VecFree(&v);
    }
}

// Print the ImgKMeansClusters 'that' on the stream 'stream'
void IKMCPrintln(const ImgKMeansClusters* const that,
    FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (stream == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'stream' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    // Print the KMeansClusters of 'that'
    KMeansClustersPrintln(IKMCKMeansClusters(that), stream);
}

// Get the index of the cluster at position 'pos' for the
// ImgKMeansClusters 'that'
int IKMCGetId(const ImgKMeansClusters* const that,
    const VecShort2D* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (pos == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(PBImgAnalysisErr->_msg, "'pos' is null");
        PBErriCatch(PBImgAnalysisErr);
    }
#endif
    // Get the KMeansClusters input over the cell
    VecFloat* inputOverCell = IKMCKGetInputOverCell(that, pos);
    // Get the index of the cluster for this pixel
    int id = KMeansClustersGetId(IKMCKMeansClusters(that), inputOverCell);
    // Free memory
    VecFree(&inputOverCell);
    // Return the id
    return id;
}

// Get the GBPixel equivalent to the cluster at position 'pos'
// for the ImgKMeansClusters 'that'
// This is the average pixel over the pixel in the cell of the cluster
GBPixel IKMCKGetPixel(const ImgKMeansClusters* const that,
    const VecShort2D* const pos) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErriTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErriCatch(PBImgAnalysisErr);
        }
        if (pos == NULL) {
            PBImgAnalysisErr->_type = PBErriTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'pos' is null");
            PBErriCatch(PBImgAnalysisErr);
        }
    #endif
    // Declare the result pixel
    GBPixel pix;
    // Get the id of the cluster for the input pixel
    int id = IKMCKGetId(that, pos);
    // Get the 'id'-th cluster's center
    const VecFloat* center =
        KMeansClustersCenter(IKMCKMeansClusters(that), id);
    // Declare a variable to calculate the average pixel
    VecFloat* avgPix = VecFloatCreate(4);
    // Calculate the average pixel
    for (int i = 0; i < VecGetDim(center); i += 4) {
        for (int j = 4; j--;) {
            VecSet(avgPix, j, VecGet(avgPix, j) + VecGet(center, i + j));
        }
    }
    VecScale(avgPix, 1.0 / round((float)VecGetDim(center) / 4.0));
    // Update the returned pixel values and ensure the converted value
    // from float to char is valid
    for (int i = 4; i--;) {
        float v = VecGet(avgPix, i);
        if (v < 0.0)
            v = 0.0;
        else if (v > 255.0)
            v = 255.0;
        pix._rgba[i] = (unsigned char)v;
    }
    // Free memory
    VecFree(&avgPix);
    // Return the result pixel
    return pix;
}

```

```

// Convert the image of the ImageKMeansClusters 'that' to its clustered
// version
// IKMCSearch must have been called previously
void IKMCcluster(const ImgKMeansClusters* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // Get the dimension of the image
    VecShort2D dim = GBGetDim(IKMCImg(that));
    // Loop on pixels
    VecShort2D pos = VecShortCreateStatic2D();
    do {
        // Get the clustered pixel for this pixel
        GBPixel clustered = IKMCGetPixel(that, &pos);
        // Replace the original pixel
        GBSetFinalPixel((GenBrush*)IKMCImg(that), &pos, &clustered);
    } while (VecStep(&pos, &dim));
}

// Get the input values for the pixel at position 'pos' according to
// the cell size of the ImgKMeansClusters 'that'
// The return is a VecFloat made of the sizeCell^2 pixels' value
// around pos ordered by ((r*256+g)*256+b)*256+a)
VecFloat* IKMCGetInputOverCell(const ImgKMeansClusters* const that,
    const VecShort2D* const pos) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (pos == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'pos' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // Create two vectors to loop on the cell
    VecShort2D from = VecShortCreateStatic2D();
    VecSet(&from, 0, -that->_size);
    VecSet(&from, 1, -that->_size);
    VecShort2D to = VecShortCreateStatic2D();
    VecSet(&to, 0, that->_size + 1);
    VecSet(&to, 1, that->_size + 1);
    // Get the pixel at the center of the cell, will be used as default
    // if the cell goes over the border of the image
    const GBPixel* defaultPixel = GBFinalPixel(IKMCImg(that), pos);
    // Declare a set to memorize the pixels in the cell
    GSet pixels = GSetCreateStatic();
    // Loop over the pixels of the cell
    VecShort2D posCell = from;
    VecShort2D posImg = VecShortCreateStatic2D();
    do {
        // If the position in the cell is inside the radius of the cell
        VecFloat2D posCellFloat = VecShortToFloat2D(&posCell);
        if ((int)round(VecNorm(&posCellFloat)) <= that->_size) {
            // Get the position in the image

```

```

    posImg = VecGetOp(pos, 1, &posCell, 1);
    // Get the pixel at this position
    const GBPixel* pix = GBFinalPixelSafe(IKMCImg(that), &posImg);
    if (pix == NULL)
        pix = defaultPixel;
    // Get the value to sort this pixel
    float valPix = 0.0;
    for (int iRgba = 4; iRgba--;)
        valPix += 256.0 * valPix + (float)(pix->_rgba[iRgba]);
    // Add the pixel to the set of pixels in the cell
    GSetAddSort(&pixels, pix, valPix);
}
} while (VecShiftStep(&posCell, &from, &to));
// Declare the result vector
VecFloat* res = VecFloatCreate(GSetNbElem(&pixels) * 4);
// Loop over the sorted pixels of the cell
int iPix = 0;
while (GSetNbElem(&pixels)) {
    const GBPixel* pix = GSetDrop(&pixels);
    // Set the result value
    for (int i = 0; i < 4; ++i)
        VecSet(res, iPix * 4 + i, (float)(pix->_rgba[i]));
    ++iPix;
}
// Return the result
return res;
}

// Load the IKMC 'that' from the stream 'stream'
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCLoad(ImgKMeansClusters* that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (stream == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'stream' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
}
#endif
// Declare a json to load the encoded data
JSONNode* json = JSONCreate();
// Load the whole encoded data
if (!JSONLoad(json, stream)) {
    return false;
}
// Decode the data from the JSON
if (!IKMCDecodeAsJSON(that, json)) {
    return false;
}
// Free the memory used by the JSON
JSONFree(&json);
// Return success code
return true;
}

// Save the IKMC 'that' to the stream 'stream'
// If 'compact' equals true it saves in compact form, else it saves in

```

```

// readable form
// There is no associated GenBrush object saved
// Return true upon success else false
bool IKMCSave(const ImgKMeansClusters* const that,
FILE* const stream, const bool compact) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (stream == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'stream' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Get the JSON encoding
    JSONNode* json = IKMCEncodeAsJSON(that);
    // Save the JSON
    if (!JSONSave(json, stream, compact)) {
        return false;
    }
    // Free memory
    JSONFree(&json);
    // Return success code
    return true;
}

// Function which return the JSON encoding of 'that'
JSONNode* IKMCEncodeAsJSON(const ImgKMeansClusters* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Create the JSON structure
    JSONNode* json = JSONCreate();
    // Declare a buffer to convert value into string
    char val[100];
    // Encode the size
    sprintf(val, "%d", that->_size);
    JSONAddProp(json, "_size", val);
    // Encode the KMeansClusters
    JSONAddProp(json, "_clusters",
        KMeansClustersEncodeAsJSON(IKMCKMeansClusters(that)));
    // Return the created JSON
    return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool IKMCDecodeAsJSON(ImgKMeansClusters* that,
const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (json == NULL) {

```

```

        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'json' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // Free the memory eventually used by the IKMC
    ImgKMeansClustersFreeStatic(that);
    // Get the size from the JSON
    JSONNode* prop = JSONProperty(json, "_size");
    if (prop == NULL) {
        return false;
    }
    that->_size = atoi(JSONLabel(JSONValue(prop, 0)));
    if (that->_size < 0) {
        return false;
    }
    // Decode the KMeansClusters
    prop = JSONProperty(json, "_clusters");
    if (!KMeansClustersDecodeAsJSON(
        (KMeansClusters*)IKMCKMeansClusters(that), prop)) {
        return false;
    }
    // Return the success code
    return true;
}

// ----- ImgSegmentor -----

// ===== Functions implementation =====

// Function which return the JSON encoding the node 'that' in the
// GenTree of criteria of a ImgSegmentor
JSONNode* ISEncodeNodeAsJSON(const GenTree* const that);

// Function which return the JSON encoding of 'that'
JSONNode* ISEncodeAsJSON(const ImgSegmentor* const that);

// Function which decode from JSON encoding 'json' to 'that'
bool ISDecodeAsJSON(ImgSegmentor* that,
    const JSONNode* const json);

// Function which decodes the JSON encoding of the
// GenTree of criteria of the ImgSegmentor 'that'
bool ISDecodeNodeAsJSON(GenTree* const that,
    const JSONNode* const json);

// Function which return the JSON encoding of 'that'
JSONNode* ISCEncodeAsJSON(
    const ImgSegmentorCriterion* const that);

// Function which return the JSON encoding of 'that'
void ISCRGBEncodeAsJSON(const ImgSegmentorCriterionRGB* const that,
    JSONNode* const json);

// Function which return the JSON encoding of 'that'
void ISCRGB2HSVEncodeAsJSON(
    const ImgSegmentorCriterionRGB2HSV* const that, JSONNode* const json);

// Function which decodes the JSON encoding of a ImgSegmentorCriterion
bool ISDecodeAsJSON(
    ImgSegmentorCriterion** const that, const JSONNode* const json);

```

```

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionRGB
bool ISCRGBDecodeAsJSON(
    ImgSegmentorCriterionRGB** const that, const JSONNode* const json);

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionRGB2HSV
bool ISCRGB2HSVDecodeAsJSON(
    ImgSegmentorCriterionRGB2HSV** const that, const JSONNode* const json);

// Function which return the JSON encoding of 'that'
void ISCDustEncodeAsJSON(const ImgSegmentorCriterionDust* const that,
    JSONNode* const json);

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionDust
bool ISCDustDecodeAsJSON(
    ImgSegmentorCriterionDust** const that, const JSONNode* const json);

// Function which return the JSON encoding of 'that'
void ISCTexEncodeAsJSON(const ImgSegmentorCriterionTex* const that,
    JSONNode* const json);

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionTex
bool ISCTexDecodeAsJSON(
    ImgSegmentorCriterionTex** const that, const JSONNode* const json);

// ===== Functions implementation =====

// Create a new static ImgSegmentor with 'nbClass' output
ImgSegmentor ImgSegmentorCreateStatic(int nbClass) {
#ifdef BUILDMODE == 0
    if (nbClass <= 0) {
        PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBImpAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
            nbClass);
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // Declare the new ImgSegmentor
    ImgSegmentor that;
    // Init properties
    that._nbClass = nbClass;
    that._criteria = GenTreeCreateStatic();
    that._flagBinaryResult = false;
    that._thresholdBinaryResult = 0.5;
    that._nbEpoch = 1;
    that._sizePool = GENALG_NBENTITIES;
    that._sizeMinPool = that._sizePool;
    that._sizeMaxPool = that._sizePool;
    that._nbElite = GENALG_NBELITES;
    that._targetBestValue = 0.9999;
    that._flagTextOMeter = false;
    that._textOMeter = NULL;
    sprintf(that._line1, IS_TRAINTXTOMETER_LINE1);
    sprintf(that._line2, IS_EVALTXTOMETER_LINE1);
    // Return the new ImgSegmentor
    return that;
}

// Create a new ImgSegmentor with 'nbClass' output

```

```

ImgSegmentor* ImgSegmentorCreate(int nbClass) {
#ifdef BUILDMODE == 0
    if (nbClass <= 0) {
        PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBIImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
            nbClass);
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Declare the new ImgSegmentor
    ImgSegmentor* that = PBErrMalloc(PBIImgAnalysisErr,
        sizeof(ImgSegmentor));
    // Init properties
    that->_nbClass = nbClass;
    that->_criteria = GenTreeCreateStatic();
    that->_flagBinaryResult = false;
    that->_thresholdBinaryResult = 0.5;
    that->_nbEpoch = 1;
    that->_sizePool = GENALG_NBENTITIES;
    that->_sizeMinPool = that->_sizePool;
    that->_sizeMaxPool = that->_sizePool;
    that->_nbElite = GENALG_NBELITES;
    that->_targetBestValue = 0.9999;
    that->_flagTextOMeter = false;
    that->_textOMeter = NULL;
    sprintf(that->_line1, IS_TRAINTXTOMETER_LINE1);
    sprintf(that->_line2, IS_EVALTXTOMETER_LINE1);
    // Return the new ImgSegmentor
    return that;
}

// Free the memory used by the static ImgSegmentor 'that'
void ImgSegmentorFreeStatic(ImgSegmentor* that) {
    if (that == NULL)
        return;
    if (that->_textOMeter != NULL)
        TextOMeterFree(&(that->_textOMeter));
    if (!GenTreeIsLeaf(ISCriteria(that))) {
        GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(ISCriteria(that));
        do {
            ImgSegmentorCriterion* criterion = GenTreeIterGetData(&iter);
            switch (criterion->_type) {
                case ISCType_RGB:
                    ImgSegmentorCriterionRGBFree(
                        (ImgSegmentorCriterionRGB**)&criterion);
                    break;
                case ISCType_RGB2HSV:
                    ImgSegmentorCriterionRGB2HSVFree(
                        (ImgSegmentorCriterionRGB2HSV**)&criterion);
                    break;
                case ISCType_Dust:
                    ImgSegmentorCriterionDustFree(
                        (ImgSegmentorCriterionDust**)&criterion);
                    break;
                case ISCType_Tex:
                    ImgSegmentorCriterionTexFree(
                        (ImgSegmentorCriterionTex**)&criterion);
                    break;
                default:
                    PBIImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
                    sprintf(PBIImgAnalysisErr->_msg,
                        "Not yet implemented type of criterion");
            }
        } while (GenTreeIterDepthNext(&iter));
    }
}

```



```

        PBErCatch(PBImgAnalysisErr);
        break;
    }
} while (GenTreeIterStep(&iter));
GenTreeIterFreeStatic(&iter);
}
GenTreeFreeStatic((GenTree*)ISCriteria(that));
}

// Free the memory used by the ImgSegmentor 'that'
void ImgSegmentorFree(ImgSegmentor** that) {
    if (that == NULL || *that == NULL)
        return;
    ImgSegmentorFreeStatic(*that);
    free(*that);
    *that = NULL;
}

// Make a prediction on the GenBrush 'img' with the ImgSegmentor 'that'
// Return an array of pointer to GenBrush, one per output class, in
// greyscale, where the color of each pixel indicates the detection of
// the corresponding class at the given pixel, white equals no
// detection, black equals detection, 50% grey equals "don't know"
GenBrush** ISPredict(const ImgSegmentor* const that,
    const GenBrush* const img) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
    if (img == NULL) {
        PBImgAnalysisErr->_type = PBErTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'img' is null");
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    // Get the dimension of the input image
    VecShort2D dim = GBGetDim(img);
    // Calculate the area of the image
    long area = VecGet(&dim, 0) * VecGet(&dim, 1);
    // Create a temporary vector to convert the image into the input
    // of a criterion
    VecFloat* input = VecFloatCreate(area * 3);
    // Declare a vector to loop on position in the image
    VecShort2D pos = VecShortCreateStatic2D();
    // Convert the image's pixels into the input VecFloat
    do {
        GBPixel pix = GBGetFinalPixel(img, &pos);
        long iPos = GBPosIndex(&pos, &dim);
        for (int iRGB = 3; iRGB--;)
            VecSet(input, iPos * 3 + iRGB, (float)(pix._rgba[iRGB]) / 255.0);
    } while (VecStep(&pos, &dim));
    // Declare a set to memorize the temporary inputs while moving
    // through the tree of criteria
    GSet inputs = GSetCreateStatic();
    // Add the initial input to the set
    GSetAppend(&inputs, input);
    // Create a set to memorize the prediction of each leaf criterion
    GSet leafPred = GSetCreateStatic();
    // Loop on criteria
    GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(ISCriteria(that));

```

```

do {
    // Get the criteria
    ImgSegmentorCriterion* criterion = GenTreeIterGetData(&iter);
    // Get the input on which to apply the criteria, this is the last
    // pushed input
    VecFloat* curInput = GSetTail(&inputs);
    // Do the prediction
    VecFloat* pred = ISCPredict(criterion, curInput, &dim);
    // If this criterion is a leaf in the tree of criteria
    if (GenTreeIsLeaf(GenTreeIterGetGenTree(&iter))) {
        // Add the result of the prediction to the set of final prediction
        GSetAppend(&leafPred, pred);
        // If the criterion is a last brother
        if (GenTreeIsLastBrother(GenTreeIterGetGenTree(&iter))) {
            // Drop and free the intermediate input
            (void)GSetDrop(&inputs);
            VecFree(&curInput);
            // In case the parent was the last brother it will be skipped
            // back by the GenTreeIterDepth and we need to drop its input
            // right away
            GenTree* parent = GenTreeParent(GenTreeIterGetGenTree(&iter));
            while (parent != NULL && GenTreeIsLastBrother(parent)) {
                curInput = GSetDrop(&inputs);
                VecFree(&curInput);
                parent = GenTreeParent(parent);
            }
        }
        // Else the criterion is a node in the tree
    } else {
        // Append the result of prediction to the intermediate input
        GSetAppend(&inputs, pred);
    }
} while(GenTreeIterStep(&iter));
GenTreeIterFreeStatic(&iter);
// Create temporary vectors to memorize the combined predictions
VecFloat* combPred = VecFloatCreate(area * ISGetNbClass(that));
VecFloat* finalPred = VecFloatCreate(area * ISGetNbClass(that));
// Combine the predictions over criteria
// The combination is the weighted average of prediction over criteria
// where the weight is the absolute value of the prediction
for (long i = area * (long)ISGetNbClass(that); i--;) {
    float sumWeight = 0.0;
    GSetIterForward iter = GSetIterForwardCreateStatic(&leafPred);
    do {
        VecFloat* pred = GSetIterGet(&iter);
        float v = VecGet(pred, i);
        VecSetAdd(combPred, i, v * fabs(v));
        sumWeight += fabs(v);
    } while (GSetIterStep(&iter));
    if (sumWeight > PBMath_EPSILON)
        VecSet(combPred, i, VecGet(combPred, i) / sumWeight);
    else
        VecSet(combPred, i, 0.0);
}
// Combine the predictions over classes
// The combination is calculated as follow:
// finalPred(i) = (pred(i)*abs(combPred(i) - sum_{j!=i}
//   combPred(j)*abs(combPred(j)) / (sum_i abs(combPred(i))
VecSetNull(&pos);
do {
    for (long iClass = ISGetNbClass(that); iClass--;) {
        float sumWeight = 0.0;

```

```

    long iPos = GBPosIndex(&pos, &dim) * ISGetNbClass(that) + iClass;
    for (long jClass = ISGetNbClass(that); jClass--;) {
        long jPos = GBPosIndex(&pos, &dim) * ISGetNbClass(that) + jClass;
        float v = VecGet(combPred, jPos);
        if (iClass == jClass) {
            VecSetAdd(finalPred, iPos, v * fabs(v));
        } else {
            VecSetAdd(finalPred, iPos, -1.0 * v * fabs(v));
        }
        sumWeight += fabs(v);
    }
    if (sumWeight > PBMATH_EPSILON)
        VecSet(finalPred, iPos, VecGet(finalPred, iPos) / sumWeight);
    else
        VecSet(finalPred, iPos, 0.0);
}
} while(VecStep(&pos, &dim));
// Allocate memory for the results
GenBrush** res = PBErrMalloc(PBImgAnalysisErr,
    sizeof(GenBrush*) * ISGetNbClass(that));
// Declare a variable to convert the prediction into pixel
GBPixel pix = GBColorWhite;
// Loop on classes
for (int iClass = ISGetNbClass(that); iClass--;) {
    // Create the result GenBrush
    res[iClass] = GBCreateImage(&dim);
    // Loop on position in the image
    VecSetNull(&pos);
    do {
        // Get the prediction value for this class and this position
        // and convert it to rgb value
        long iPos = GBPosIndex(&pos, &dim);
        float p = VecGet(finalPred, iPos * ISGetNbClass(that) + iClass);
        if (ISGetFlagBinaryResult(that)) {
            if (p > ISGetThresholdBinaryResult(that))
                p = 1.0;
            else
                p = -1.0;
        }
        unsigned char pChar = 255 -
            (unsigned char)round(255.0 * (p * 0.5 + 0.5));
        // Convert the prediction to a pixel
        pix._rgba[GBPixelRed] = pix._rgba[GBPixelGreen] =
            pix._rgba[GBPixelBlue] = pChar;
        // Set the pixel in the result image
        GBSetFinalPixel(res[iClass], &pos, &pix);
    } while (VecStep(&pos, &dim));
}
// Free memory
while (GSetNbElem(&leafPred) > 0) {
    VecFloat* pred = GSetPop(&leafPred);
    VecFree(&pred);
}
do {
    VecFloat* curInput = GSetDrop(&inputs);
    VecFree(&curInput);
} while (GSetNbElem(&inputs) > 0);
VecFree(&finalPred);
VecFree(&combPred);
// Return the result
return res;
}

```

```

// Handler for the signal Ctrl-C
void ISTrainHandlerCtrlC(int sig) {
    (void)sig;
    PBIA_CtrlC = true;
    time_t intTime = time(NULL);
    char* strIntTime = ctime(&intTime);
    printf("\n%s!!! ISTrain Interrupted by Ctrl-C !!!\n", strIntTime);
    fflush(stdout);
}

// Train the ImageSegmentor 'that' on the data set 'dataSet' using
// the data of the first category in 'dataSet'. If the data set has a
// second category it will be used for validation
// random must have been called before calling ISTrain
void ISTrain(ImgSegmentor* const that,
    const GDataSetGenBrushPair* const dataset) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (dataset == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'dataset' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (ISGetNbClass(that) > GDSGetNbMask(dataset)) {
            PBIImgAnalysisErr->_type = PBErrTypeInvalidData;
            sprintf(PBIImgAnalysisErr->_msg,
                "Not enough masks in the dataset (%d<=%d)",
                ISGetNbClass(that), GDSGetNbMask(dataset));
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Set the handler to catch the signal Ctrl-C
    signal(SIGINT, ISTrainHandlerCtrlC);
    // If there is no criterion, nothing to do
    if (ISGetNbCriterion(that) == 0)
        return;
    // Memorize the current flag for binarization of results
    bool curFlagBinary = ISGetFlagBinaryResult(that);
    // Turn on the binarization
    ISSetFlagBinaryResult(that, true);
    // Create two vectors to memorize the number of int and float
    // parameters for each criterion
    VecLong* nbParamInt = VecLongCreate(ISGetNbCriterion(that));
    VecLong* nbParamFloat = VecLongCreate(ISGetNbCriterion(that));
    // Declare two variables to memorize the total number of int and
    // float parameters
    long nbTotalParamInt = 0;
    long nbTotalParamFloat = 0;
    // Get the number of int and float parameters for each criterion
    int iCrit = 0;
    GenTreeIterDepth iter = GenTreeIterDepthCreateStatic(ISCriteria(that));
    do {
        ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
        long nb = ISGetNbParamInt(crit);
        VecSet(nbParamInt, iCrit, nb);
        nbTotalParamInt += nb;
        nb = ISGetNbParamFloat(crit);
    }

```

```

    VecSet(nbParamFloat, iCrit, nb);
    nbTotalParamFloat += nb;
    ++iCrit;
} while (GenTreeIterStep(&iter));
char cpFilename[200] = {'\0'};
// If there are parameters
if (nbTotalParamInt > 0 || nbTotalParamFloat > 0) {
    // Create the GenAlg to search parameters' value
    GenAlg* ga = GenAlgCreate(ISGetSizePool(that), ISGetNbElite(that),
        nbTotalParamFloat, nbTotalParamInt);
    // Set the min and max size of the pool
    GASetNbMaxAdn(ga, ISGetSizeMaxPool(that));
    GASetNbMinAdn(ga, ISGetSizeMinPool(that));
    // Loop on the criterion to initialise the parameters bound
    GenTreeIterReset(&iter);
    long shiftParamInt = 0;
    long shiftParamFloat = 0;
    do {
        ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
        ISCSetsAdnInt(crit, ga, shiftParamInt);
        shiftParamInt += ISCSGetNbParamInt(crit);
        ISCSetsAdnFloat(crit, ga, shiftParamFloat);
        shiftParamFloat += ISCSGetNbParamFloat(crit);
    } while (GenTreeIterStep(&iter));
    // Initialise the GenAlg
    GAInit(ga);
    // Set the TextOMeter flag of the GenAlg same as the one of the
    // ImgSegmentor
    GASetTextOMeterFlag(ga, ISGetFlagTextOMeter(that));
    // Declare a variable to memorize the current best value
    float bestValue = 0.0;
    // Loop over epochs
    do {
        // Loop over the GenAlg entities
        for (int iEnt = 0; iEnt < GAGetNbAdns(ga) &&
            bestValue < ISGetTargetBestValue(that) && !PBIA_CtrlC; ++iEnt) {
            // If this entity is a new one
            if (GAAdnIsNew(GAAdn(ga, iEnt))) {
                // Loop on the criterion to set the criteria parameters with
                // this entity's adn
                GenTreeIterReset(&iter);
                shiftParamInt = 0;
                shiftParamFloat = 0;
                do {
                    ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
                    ISCSetsAdnInt(crit, GAAdn(ga, iEnt), shiftParamInt);
                    shiftParamInt += ISCSGetNbParamInt(crit);
                    ISCSetsAdnFloat(crit, GAAdn(ga, iEnt), shiftParamFloat);
                    shiftParamFloat += ISCSGetNbParamFloat(crit);
                } while (GenTreeIterStep(&iter));
                // Update the info for the TexOMeter
                if (ISGetFlagTextOMeter(that)) {
                    sprintf(that->_line1, IS_TRAINTXTOMETER_FORMAT1,
                        GAGetCurEpoch(ga) + 1L, (long int)ISGetNbEpoch(that),
                        iEnt + 1, GAGetNbAdns(ga));
                }
                // Evaluate the ImgSegmentor for this entity's adn on the
                // dataset
                const int iCatTraining = 0;
                float value = ISEvaluate(that, dataset, iCatTraining);
                // Update the value of this entity's adn
                GASetAdnValue(ga, GAAdn(ga, iEnt), value);
            }
        }
    }
}

```

```

// If the value is the best value
if (value - bestValue > PBMath_EPSILON) {
    bestValue = value;
    printf("Epoch %05ld/%05u ",
        GAGetCurEpoch(ga) + 1, ISGetNbEpoch(that));
    printf("TrainAcc[0,1] %f/%f ", bestValue,
        ISGetTargetBestValue(that));
    // If the dataset has an evaluation category
    float evalValue = 0.0;
    if (GDSGetNbCat(dataset) > 1) {
        // Evaluate the new best entity on the validation category
        const int iCatValid = 1;
        evalValue = ISEvaluate(that, dataset, iCatValid);
        printf("EvalAcc[0,1] %f ", evalValue);
    }
    time_t improvTime = time(NULL);
    char* strImprovTime = ctime(&improvTime);
    printf("on %s", strImprovTime);
    fflush(stdout);
    // Save the ImgSegmentor
    if (GDSGetNbCat(dataset) > 1) {
        sprintf(cpFilename, "%05ld_%f_%f-" IS_CHECKPOINTFILENAME, GAGetCurEpoch(ga) + 1L, bestValue, evalValue);
    } else {
        sprintf(cpFilename, "%05ld_%f-" IS_CHECKPOINTFILENAME, GAGetCurEpoch(ga) + 1L, bestValue);
    }
    FILE* fpCheckpoint = fopen(cpFilename, "w");
    if (!ISSave(that, fpCheckpoint, false)) {
        fprintf(stderr, "Couldn't save the checkpoint %s\n",
            cpFilename);
    }
    fclose(fpCheckpoint);
}
}
}
// Step the GenAlg
GAStep(ga);
} while (GAGetCurEpoch(ga) < ISGetNbEpoch(that) &&
    bestValue < ISGetTargetBestValue(that) && !PBIA_CtrlC);
// Loop on the criterion to set the criteria to the best one
GenTreeIterReset(&iter);
shiftParamInt = 0;
shiftParamFloat = 0;
do {
    ImgSegmentorCriterion* crit = GenTreeIterGetData(&iter);
    ISCSetsAdnInt(crit, GABestAdn(ga), shiftParamInt);
    shiftParamInt += ISCSetsNbParamInt(crit);
    ISCSetsAdnFloat(crit, GABestAdn(ga), shiftParamFloat);
    shiftParamFloat += ISCSetsNbParamFloat(crit);
} while (GenTreeIterStep(&iter));
// Free memory
GenAlgFree(&ga);
}
// Reload the checkpoint at the end of the training to
// return the ImgSegmentor in its best version
FILE* fpCheckpoint = fopen(cpFilename, "r");
if (!fpCheckpoint) {
    if (!ISLoad(that, fpCheckpoint)) {
        fprintf(stderr, "Couldn't reload the checkpoint %s\n",
            cpFilename);
    }
}
fclose(fpCheckpoint);
}

```

```

// Free memory
GenTreeIterFreeStatic(&iter);
VecFree(&nbParamInt);
VecFree(&nbParamFloat);
// Put back the flag for binarization in its original state
ISSetFlagBinaryResult(that, curFlagBinary);
// Reset the signal handler for the signal Ctrl-C to its default
signal(SIGINT, SIG_DFL);
}

// Evaluate the ImageSegmentor 'that' on the data set 'dataSet' using
// the data of the 'iCat' category in 'dataSet'
// random must have been called before calling ISTrain
// Return a value in [0.0, 1.0], 0.0 being worst and 1.0 being best
float ISEvaluate(ImageSegmentor* const that,
const GDataSetGenBrushPair* const dataset, const int iCat) {
// Declare a variable to memorize the result value
float value = 0.0;
// Declare a variable to memorize the color of the mask
const GBPixel rgbaMask = GBColorBlack;
// Reset the iterator of the GDataSet
GDSReset(dataset, iCat);
// Loop on the samples
long iSample = 0;
do {
// Update the info for the TexOMeter and refresh it
if (ISGetFlagTextOMeter(that)) {
sprintf(that->_line2, IS_EVALTXTOMETER_FORMAT1,
iSample, GDSGetSizeCat(dataset, iCat));
ISUpdateTextOMeter(that);
}
// Get the next sample
GDSGenBrushPair* sample = GDSGetSample(dataset, iCat);
// Do the prediction on the sample
GenBrush** pred = ISPredict(that, sample->_img);
// Check the prediction against the masks
float valMask = 0.0;
for (int iMask = ISGetNbClass(that); iMask--;) {
valMask += IntersectionOverUnion(
sample->_mask[iMask], pred[iMask], &rgbaMask);
}
value += valMask / (float)GDSGetNbMask(dataset);
// Free memory
for (int iClass = ISGetNbClass(that); iClass--;)
GBFree(pred + iClass);
free(pred);
GDSGenBrushPairFree(&sample);
++iSample;
} while (GDSStepSample(dataset, iCat) && !PBIA_CtrlC);
// Get the average value over all samples
value /= (float)GDSGetSizeCat(dataset, iCat);
// Return the result of the evaluation
return value;
}

// Set the flag memorizing if the TextOMeter is displayed for
// the ImageSegmentor 'that' to 'flag'
void ISSetFlagTextOMeter(ImageSegmentor* const that, bool flag) {
#if BUILDMODE == 0
if (that == NULL) {
PBImpAnalysisErr->_type = PBErrTypeNullPointer;
sprintf(PBImpAnalysisErr->_msg, "'that' is null");
}

```

```

        PBErriCatch(PBImgAnalysisErr);
    }
#endif
// If the requested flag is different from the current flag;
if (that->_flagTextOMeter != flag) {
    if (flag && that->_textOMeter == NULL) {
        char title[] = "ImgSegmentor";
        int width = strlen(IS_TRAINTXTOMETER_LINE1) + 1;
        int height = 3;
        that->_textOMeter = TextOMeterCreate(title, width, height);
    }
    if (!flag && that->_textOMeter != NULL) {
        TextOMeterFree(&(that->_textOMeter));
    }
    that->_flagTextOMeter = flag;
}
}

// Refresh the content of the TextOMeter attached to the
// ImgSegmentor 'that'
void ISUpdateTextOMeter(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErriTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErriCatch(PBImgAnalysisErr);
    }
    if (that->_textOMeter == NULL) {
        PBImgAnalysisErr->_type = PBErriTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that->_textOMeter' is null");
        PBErriCatch(PBImgAnalysisErr);
    }
#endif
    // Clear the TextOMeter
    TextOMeterClear(that->_textOMeter);
    // .....
    TextOMeterPrint(that->_textOMeter, that->_line1);
    TextOMeterPrint(that->_textOMeter, that->_line2);
    // Flush the content of the TextOMeter
    TextOMeterFlush(that->_textOMeter);
}

// Load the ImgSegmentor from the stream
// If the ImgSegmentor is already allocated, it is freed before loading
// Return true upon success else false
bool ISLoad(ImgSegmentor* that, FILE* const stream) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErriTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErriCatch(PBImgAnalysisErr);
    }
    if (stream == NULL) {
        PBImgAnalysisErr->_type = PBErriTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'stream' is null");
        PBErriCatch(PBImgAnalysisErr);
    }
#endif
    // Declare a json to load the encoded data
    JSONNode* json = JSONCreate();
    // Load the whole encoded data
    if (!JSONLoad(json, stream)) {

```



```

        return false;
    }
    // Decode the data from the JSON
    if (!ISDecodeAsJSON(that, json)) {
        return false;
    }
    // Free the memory used by the JSON
    JSONFree(&jjson);
    // Return success code
    return true;
}

// Save the ImgSegmentor to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success else false
bool ISSave(const ImgSegmentor* const that,
            FILE* const stream, const bool compact) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (stream == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'stream' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Get the JSON encoding
    JSONNode* json = ISEncodeAsJSON(that);
    // Save the JSON
    if (!JSONSave(json, stream, compact)) {
        return false;
    }
    // Free memory
    JSONFree(&jjson);
    // Return success code
    return true;
}

// Function which return the JSON encoding of 'that'
JSONNode* ISEncodeAsJSON(const ImgSegmentor* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the JSON structure
    JSONNode* json = JSONCreate();
    // Declare a buffer to convert value into string
    char val[100];
    // Number of segmentation class
    sprintf(val, "%d", that->_nbClass);
    JSONAddProp(json, "_nbClass", val);
    // Flag to apply or not the binarization
    sprintf(val, "%d", that->_flagBinaryResult);
    JSONAddProp(json, "_flagBinaryResult", val);
    // Threshold value for the binarization of result of prediction

```

```

    sprintf(val, "%f", that->_thresholdBinaryResult);
    JSONAddProp(json, "_thresholdBinaryResult", val);
    // Nb of epoch
    sprintf(val, "%u", that->_nbEpoch);
    JSONAddProp(json, "_nbEpoch", val);
    // Size pool for training
    sprintf(val, "%d", that->_sizePool);
    JSONAddProp(json, "_sizePool", val);
    // Nb elite for training
    sprintf(val, "%d", that->_nbElite);
    JSONAddProp(json, "_nbElite", val);
    // Threshold to stop the training once
    sprintf(val, "%f", that->_targetBestValue);
    JSONAddProp(json, "_targetBestValue", val);
    // Tree of criterion
    JSONAddProp(json, "_criteria",
        ISEncodeNodeAsJSON(ISCriteria(that)));
    // Return the created JSON
    return json;
}

// Function which return the JSON encoding the node 'that' in the
// GenTree of criteria of a ImgSegmentor
JSONNode* ISEncodeNodeAsJSON(const GenTree* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBMathErr->_type = PBErrTypeNullPointer;
            sprintf(PBMathErr->_msg, "'that' is null");
            PBErrCatch(PBMathErr);
        }
    #endif
    // Create the JSON structure
    JSONNode* json = JSONCreate();
    // If there is a criterion on the node
    if (GenTreeData(that) != NULL) {
        // Encode the criterion
        JSONAddProp(json, "_criterion",
            ISEncodeNodeAsJSON(
                (ImgSegmentorCriterion*)GenTreeData(that)));
    }
    // Add the number of subtrees
    char val[100];
    sprintf(val, "%ld", GSetNbElem(&(that->_subtrees)));
    JSONAddProp(json, "_nbSubtree", val);
    // If there are subtrees
    if (!GenTreeIsLeaf(that)) {
        // Loop on the subtrees
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GenTreeSubtrees(that));
        int iSubtree = 0;
        do {
            GenTree* subtree = GSetIterGet(&iter);
            // Add the subtree
            char lblSubtree[100];
            sprintf(lblSubtree, "_subtree_%d", iSubtree);
            JSONAddProp(json, lblSubtree,
                ISEncodeNodeAsJSON(subtree));
            ++iSubtree;
        } while (GSetIterStep(&iter));
    }
    // Return the created JSON
    return json;
}

```

```

}

// Function which decode from JSON encoding 'json' to 'that'
bool ISDecodeAsJSON(ImgSegmentor* that,
    const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (json == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'json' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // If 'that' is already allocated
    if (that != NULL)
        // Free memory
        ImgSegmentorFreeStatic(that);
    // Get the nb of class from the JSON
    JSONNode* prop = JSONProperty(json, "_nbClass");
    if (prop == NULL) {
        return false;
    }
    int nbClass = atoi(JSONLabel(JSONValue(prop, 0)));
    // If data are invalid
    if (nbClass <= 0)
        return false;
    // Allocate memory
    *that = ImgSegmentorCreateStatic(nbClass);
    // Flag to apply or not the binarization
    prop = JSONProperty(json, "_flagBinaryResult");
    if (prop == NULL) {
        return false;
    }
    int flagBinaryResult = atoi(JSONLabel(JSONValue(prop, 0)));
    if (flagBinaryResult == 0)
        that->_flagBinaryResult = false;
    else if (flagBinaryResult == 1)
        that->_flagBinaryResult = true;
    else
        return false;
    // Threshold value for the binarization of result of prediction
    prop = JSONProperty(json, "_thresholdBinaryResult");
    if (prop == NULL) {
        return false;
    }
    that->_thresholdBinaryResult = atof(JSONLabel(JSONValue(prop, 0)));
    // Nb of epoch
    prop = JSONProperty(json, "_nbEpoch");
    if (prop == NULL) {
        return false;
    }
    int nbEpoch = atoi(JSONLabel(JSONValue(prop, 0)));
    if (nbEpoch < 1)
        return false;
    that->_nbEpoch = (unsigned int)nbEpoch;
    // Size pool for training
    prop = JSONProperty(json, "_sizePool");
    if (prop == NULL) {

```

```

        return false;
    }
    int sizePool = atoi(JSONLabel(JSONValue(prop, 0)));
    if (sizePool < 3)
        return false;
    that->_sizePool = sizePool;
    // Nb elite for training
    prop = JSONProperty(json, "_nbElite");
    if (prop == NULL) {
        return false;
    }
    int nbElite = atoi(JSONLabel(JSONValue(prop, 0)));
    if (nbElite < 2 || nbElite > sizePool - 1)
        return false;
    that->_nbElite = nbElite;
    // Threshold to stop the training once
    prop = JSONProperty(json, "_targetBestValue");
    if (prop == NULL) {
        return false;
    }
    float targetBestValue = atof(JSONLabel(JSONValue(prop, 0)));
    if (targetBestValue < 0.0 || targetBestValue > 1.0)
        return false;
    that->_targetBestValue = targetBestValue;
    // Tree of criterion
    prop = JSONProperty(json, "_criteria");
    if (prop == NULL) {
        return false;
    }
    if (!ISDecodeNodeAsJSON(&(that->_criteria), prop)) {
        return false;
    }
    // Return the success code
    return true;
}

// Function which decodes the JSON encoding of the
// GenTree of criteria of the ImgSegmentor 'that'
bool ISDecodeNodeAsJSON(GenTree* const that,
    const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (json == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'json' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // If there is a criterion
    JSONNode* prop = JSONProperty(json, "_criterion");
    if (prop != NULL) {
        // Decode the criterion
        if (!ISDecodeAsJSON((ImgSegmentorCriterion*)&(that->_data), prop)) {
            return false;
        }
    }
    // Get the number of subtrees
    prop = JSONProperty(json, "_nbSubtree");

```

```

    if (prop == NULL) {
        return false;
    }
    int nbSubtree = atoi(JSONLabel(JSONValue(prop, 0)));
    if (nbSubtree < 0)
        return false;
    // Loop on subtree
    for (int iSubtree = 0; iSubtree < nbSubtree; ++iSubtree) {
        // Get the subtree
        char lblSubtree[100];
        sprintf(lblSubtree, "_subtree_%d", iSubtree);
        prop = JSONProperty(json, lblSubtree);
        if (prop == NULL) {
            return false;
        }
        // Decode the subtree
        GenTree* subtree = GenTreeCreate();
        if (!ISDecodeNodeAsJSON(subtree, prop)) {
            return false;
        }
        GenTreeAppendSubtree(that, subtree);
    }
    // Return the success code
    return true;
}

// Create a new static ImgSegmentorCriterion with 'nbClass' output
// and the type of criteria 'type'
ImgSegmentorCriterion ImgSegmentorCriterionCreateStatic(int nbClass,
    ISCType type) {
    #if BUILDMODE == 0
        if (nbClass <= 0) {
            PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBImpAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
                nbClass);
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    // Declare the new ImgSegmentorCriterion
    ImgSegmentorCriterion that;
    // Set the properties
    that._nbClass = nbClass;
    that._type = type;
    // Return the new ImgSegmentorCriterion
    return that;
}

// Free the memory used by the static ImgSegmentorCriterion 'that'
void ImgSegmentorCriterionFreeStatic(ImgSegmentorCriterion* that) {
    if (that == NULL)
        return;
    // Nothing to do
}

// Make the prediction on the 'input' values by calling the appropriate
// function according to the type of criteria
// 'input' 's format is width*height*3, values in [0.0, 1.0]
// Return values are width*height*nbClass, values in [-1.0, 1.0]
VecFloat* ISCPredict(const ImgSegmentorCriterion* const that,
    const VecFloat* input, const VecShort2D* const dim) {
    #if BUILDMODE == 0
        if (that == NULL) {

```

```

    PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBIImgAnalysisErr);
}
if (input == NULL) {
    PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBIImgAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBIImgAnalysisErr);
}
#endif
// Declare a variable to memorize the result
VecFloat* res = NULL;
// Call the appropriate function based on the type
switch(that->_type) {
    case ISCType_RGB:
        res = ISCRGBPredict((const ImgSegmentorCriterionRGB*)that,
            input, dim);
        break;
    case ISCType_RGB2HSV:
        res = ISCRGB2HSVPredict((const ImgSegmentorCriterionRGB2HSV*)that,
            input, dim);
        break;
    case ISCType_Dust:
        res = ISCDustPredict((const ImgSegmentorCriterionDust*)that,
            input, dim);
        break;
    case ISCType_Tex:
        res = ISCTexPredict((const ImgSegmentorCriterionTex*)that,
            input, dim);
        break;
    default:
        PBIImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
        sprintf(PBIImgAnalysisErr->_msg,
            "Not yet implemented type of criterion");
        PBErrCatch(PBIImgAnalysisErr);
        break;
}
// Return the result
return res;
}

JSONNode* ISCEncodeAsJSON(
    const ImgSegmentorCriterion* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Declare a variable to memorize the result
    JSONNode* json = JSONCreate();
    // Declare a buffer to convert value into string
    char val[100];
    // Type
    sprintf(val, "%d", that->_type);
    JSONAddProp(json, "_type", val);
    // Number of segmentation class
    sprintf(val, "%d", that->_nbClass);
    JSONAddProp(json, "_nbClass", val);
    // Call the appropriate function based on the type
    switch(that->_type) {

```

```

        case ISCType_RGB:
            ISCRGBEncodeAsJSON((const ImgSegmentorCriterionRGB*)that, json);
            break;
        case ISCType_RGB2HSV:
            ISCRGB2HSVEncodeAsJSON(
                (const ImgSegmentorCriterionRGB2HSV*)that, json);
            break;
        case ISCType_Dust:
            ISCDustEncodeAsJSON(
                (const ImgSegmentorCriterionDust*)that, json);
            break;
        case ISCType_Tex:
            ISCTexEncodeAsJSON(
                (const ImgSegmentorCriterionTex*)that, json);
            break;
        default:
            PBImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
            sprintf(PBImgAnalysisErr->_msg,
                "Not yet implemented type of criterion");
            PBErrCatch(PBImgAnalysisErr);
            break;
    }
    // Return the result
    return json;
}

// Function which decodes the JSON encoding of a ImgSegmentorCriterion
bool ISCDecodeAsJSON(
    ImgSegmentorCriterion** const that, const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
    if (json == NULL) {
        PBImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'json' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    // Get the type of the criterion
    JSONNode* prop = JSONProperty(json, "_type");
    if (prop == NULL) {
        return false;
    }
    ISCType type = atoi(JSONLabel(JSONValue(prop, 0)));
    // Declare a variable to memorize the returned code
    bool ret = true;
    // Call the appropriate function based on the type
    switch(type) {
        case ISCType_RGB:
            ret = ISCRGBDecodeAsJSON((ImgSegmentorCriterionRGB**)that, json);
            break;
        case ISCType_RGB2HSV:
            ret = ISCRGB2HSVDecodeAsJSON(
                (ImgSegmentorCriterionRGB2HSV**)that, json);
            break;
        case ISCType_Dust:
            ret = ISCDustDecodeAsJSON((ImgSegmentorCriterionDust**)that, json);
            break;
        case ISCType_Tex:

```

```

        ret = ISCTexDecodeAsJSON((ImgSegmentorCriterionTex**)that, json);
        break;
    default:
        ret = false;
        break;
    }
    // Return the result code
    return ret;
}

// Return the number of int parameters for the criterion 'that'
long _ISCGetNbParamInt(const ImgSegmentorCriterion* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Declare a variable to memorize the result
    long res = 0;
    // Call the appropriate function based on the type
    switch(that->_type) {
        case ISCType_RGB:
            res = ISCRGBGetNbParamInt((const ImgSegmentorCriterionRGB*)that);
            break;
        case ISCType_RGB2HSV:
            res = ISCRGB2HSVGetNbParamInt(
                (const ImgSegmentorCriterionRGB2HSV*)that);
            break;
        case ISCType_Dust:
            res = ISCDustGetNbParamInt((const ImgSegmentorCriterionDust*)that);
            break;
        case ISCType_Tex:
            res = ISCTexGetNbParamInt((const ImgSegmentorCriterionTex*)that);
            break;
        default:
            PBIImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
            sprintf(PBIImgAnalysisErr->_msg,
                "Not yet implemented type of criterion");
            PBErrCatch(PBIImgAnalysisErr);
            break;
    }
    // Return the result
    return res;
}

// Return the number of float parameters for the criterion 'that'
long _ISCGetNbParamFloat(const ImgSegmentorCriterion* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Declare a variable to memorize the result
    long res = 0;
    // Call the appropriate function based on the type
    switch(that->_type) {
        case ISCType_RGB:
            res = ISCRGBGetNbParamFloat((const ImgSegmentorCriterionRGB*)that);

```



```

        break;
    case ISCType_RGB2HSV:
        res = ISCRGB2HSVGetNbParamFloat(
            (const ImgSegmentorCriterionRGB2HSV*)that);
        break;
    case ISCType_Dust:
        res = ISCDustGetNbParamFloat(
            (const ImgSegmentorCriterionDust*)that);
        break;
    case ISCType_Tex:
        res = ISCTexGetNbParamFloat((const ImgSegmentorCriterionTex*)that);
        break;
    default:
        PBImpAnalysisErr->_type = PBErrTypeNotYetImplemented;
        sprintf(PBImpAnalysisErr->_msg,
            "Not yet implemented type of criterion");
        PBErrCatch(PBImpAnalysisErr);
        break;
}
// Return the result
return res;
}

// Set the bounds of int parameters for training of the criterion 'that'
void _ISCSetBoundsAdnInt(const ImgSegmentorCriterion* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
        if (ga == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'ga' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    // Call the appropriate function based on the type
    switch(that->_type) {
        case ISCType_RGB:
            ISCRGBSetBoundsAdnInt((const ImgSegmentorCriterionRGB*)that,
                ga, shift);
            break;
        case ISCType_RGB2HSV:
            ISCRGB2HSVSetBoundsAdnInt(
                (const ImgSegmentorCriterionRGB2HSV*)that, ga, shift);
            break;
        case ISCType_Dust:
            ISCDustSetBoundsAdnInt((const ImgSegmentorCriterionDust*)that,
                ga, shift);
            break;
        case ISCType_Tex:
            ISCTexSetBoundsAdnInt((const ImgSegmentorCriterionTex*)that,
                ga, shift);
            break;
        default:
            PBImpAnalysisErr->_type = PBErrTypeNotYetImplemented;
            sprintf(PBImpAnalysisErr->_msg,
                "Not yet implemented type of criterion");
            PBErrCatch(PBImpAnalysisErr);
            break;
    }
}

```

```

    }
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void _ISCSetBoundsAdnFloat(const ImgSegmentorCriterion* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (ga == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Call the appropriate function based on the type
    switch(that->_type) {
        case ISCType_RGB:
            ISCRGBSetBoundsAdnFloat((const ImgSegmentorCriterionRGB*)that,
                ga, shift);
            break;
        case ISCType_RGB2HSV:
            ISCRGB2HSVSetBoundsAdnFloat(
                (const ImgSegmentorCriterionRGB2HSV*)that, ga, shift);
            break;
        case ISCType_Dust:
            ISCDustSetBoundsAdnFloat((const ImgSegmentorCriterionDust*)that,
                ga, shift);
            break;
        case ISCType_Tex:
            ISCTexSetBoundsAdnFloat((const ImgSegmentorCriterionTex*)that,
                ga, shift);
            break;
        default:
            PBIImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
            sprintf(PBIImgAnalysisErr->_msg,
                "Not yet implemented type of criterion");
            PBErrCatch(PBIImgAnalysisErr);
            break;
    }
}

// Set the values of int parameters for training of the criterion 'that'
void _ISCSetAdnInt(const ImgSegmentorCriterion* const that,
    const GenAlgAdn* const adn, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (adn == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Call the appropriate function based on the type

```

```

switch(that->_type) {
case ISCType_RGB:
    ISCRGBSetAdnInt((const ImgSegmentorCriterionRGB*)that,
        adn, shift);
    break;
case ISCType_RGB2HSV:
    ISCRGB2HSVSetAdnInt((const ImgSegmentorCriterionRGB2HSV*)that,
        adn, shift);
    break;
case ISCType_Dust:
    ISCDustSetAdnInt((const ImgSegmentorCriterionDust*)that,
        adn, shift);
    break;
case ISCType_Tex:
    ISCTexSetAdnInt((const ImgSegmentorCriterionTex*)that,
        adn, shift);
    break;
default:
    PBIImgAnalysisErr->_type = PBErrTypeNotYetImplemented;
    sprintf(PBIImgAnalysisErr->_msg,
        "Not yet implemented type of criterion");
    PBErrCatch(PBIImgAnalysisErr);
    break;
}
}

// Set the values of float parameters for training of the criterion
// 'that'
void _ISCSetAdnFloat(const ImgSegmentorCriterion* const that,
    const GenAlgAdn* const adn, const long shift) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (adn == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
}
#endif
// Call the appropriate function based on the type
switch(that->_type) {
case ISCType_RGB:
    ISCRGBSetAdnFloat((const ImgSegmentorCriterionRGB*)that,
        adn, shift);
    break;
case ISCType_RGB2HSV:
    ISCRGB2HSVSetAdnFloat((const ImgSegmentorCriterionRGB2HSV*)that,
        adn, shift);
    break;
case ISCType_Dust:
    ISCDustSetAdnFloat((const ImgSegmentorCriterionDust*)that,
        adn, shift);
    break;
case ISCType_Tex:
    ISCTexSetAdnFloat((const ImgSegmentorCriterionTex*)that,
        adn, shift);
    break;
default:
    PBIImgAnalysisErr->_type = PBErrTypeNotYetImplemented;

```

```

        sprintf(PBImgAnalysisErr->_msg,
            "Not yet implemented type of criterion");
        PBErrCatch(PBImgAnalysisErr);
        break;
    }
}

// ---- ImgSegmentorCriterionRGB

// Create a new ImgSegmentorCriterionRGB with 'nbClass' output
ImgSegmentorCriterionRGB* ImgSegmentorCriterionRGBCreate(
    const int nbClass) {
    #if BUILDMODE == 0
        if (nbClass <= 0) {
            PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
                nbClass);
            PBErrCatch(PBImgAnalysisErr);
        }
    #endif
    // Allocate memory for the new ImgSegmentorCriterionRGB
    ImgSegmentorCriterionRGB* that = PBErrMalloc(PBImgAnalysisErr,
        sizeof(ImgSegmentorCriterionRGB));
    // Create the parent ImgSegmentorCriterion
    that->_criterion = ImgSegmentorCriterionCreateStatic(nbClass,
        ISCType_RGB);
    // Create the NeuraNet
    const int nbInput = 3;
    const int nbHiddenPerLayer = fsquare(nbInput) * nbClass;
    const int nbHiddenLayer = 1;
    VecLong* hidden = VecLongCreate(nbHiddenLayer);
    for (int iLayer = nbHiddenLayer; iLayer--;)
        VecSet(hidden, iLayer, nbHiddenPerLayer);
    that->_nn = NeuraNetCreateFullyConnected(nbInput, nbClass, hidden);
    VecFree(&hidden);
    // Return the new ImgSegmentorCriterionRGB
    return that;
}

// Free the memory used by the ImgSegmentorCriterionRGB 'that'
void ImgSegmentorCriterionRGBFree(ImgSegmentorCriterionRGB** that) {
    if (that == NULL || *that == NULL)
        return;
    // Free memory
    ImgSegmentorCriterionFreeStatic((ImgSegmentorCriterion*)(*that));
    NeuraNetFree(&((*that)->_nn));
    free(*that);
}

// Function which return the JSON encoding of 'that'
void ISCRGBEncodeAsJSON(
    const ImgSegmentorCriterionRGB* const that, JSONNode* const json) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImgAnalysisErr);
        }
    #endif
    // NeuraNet model
    JSONAddProp(json, "_neuranet", NNEncodeAsJSON(that->_nn));
}

```

```

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionRGB
bool ISCRGBDecodeAsJSON(
    ImgSegmentorCriterionRGB** const that, const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (json == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'json' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    // If the criterion exists
    if (*that != NULL) {
        // Free the memory
        ImgSegmentorCriterionRGBFree(that);
    }
    // Get the number of class
    JSONNode* prop = JSONProperty(json, "_nbClass");
    if (prop == NULL) {
        return false;
    }
    int nbClass = atoi(JSONLabel(JSONValue(prop, 0)));
    // If the number of class is invalid
    if (nbClass < 1)
        // Return the error code
        return false;
    // Create the criterion
    *that = ImgSegmentorCriterionRGBCreate(nbClass);
    // If we couldn't create the criterion
    if (*that == NULL)
        // Return the failure code
        return false;
    // Decode the NeuraNet
    prop = JSONProperty(json, "_neuranet");
    if (prop == NULL) {
        return false;
    }
    if (!NNDecodeAsJSON(&((*that)->_nn), prop))
        return false;
    // Return the success code
    return true;
}

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionRGB that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCRGBPredict(const ImgSegmentorCriterionRGB* const that,
    const VecFloat* input, const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (input == NULL) {

```

```

    PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBIImgAnalysisErr->_msg, "'input' is null");
    PBErrCatch(PBIImgAnalysisErr);
}
if (dim == NULL) {
    PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
    sprintf(PBIImgAnalysisErr->_msg, "'dim' is null");
    PBErrCatch(PBIImgAnalysisErr);
}
if ((VecGet(dim, 0) * VecGet(dim, 1) * 3) != VecGetDim(input)) {
    PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
    sprintf(PBIImgAnalysisErr->_msg,
        "'input' 's dim is invalid (%ld=%d*d*3)", VecGetDim(input),
        VecGet(dim, 0), VecGet(dim, 1));
    PBErrCatch(PBIImgAnalysisErr);
}
#endif
/*
    printf("ISCRGB2Predict <%.3f,%.3f,%.3f %.3f,%.3f,%.3f ...>\n",
        VecGet(input, 0), VecGet(input, 1), VecGet(input, 2),
        VecGet(input, 3), VecGet(input, 4), VecGet(input, 5));
*/
// Calculate the area of the input image
long area = VecGet(dim, 0) * VecGet(dim, 1);
// Allocate memory for the result
VecFloat* res = VecFloatCreate(area * (long)ISCGetNbClass(that));
// Declare variables to memorize the input/output of the NeuraNet
VecFloat3D in = VecFloatCreateStatic3D();
VecFloat* out = VecFloatCreate(ISCGetNbClass(that));
// Apply the NeuraNet on inputs
for (long iInput = area; iInput-- && !PBIA_CtrlC;) {
    for (long i = 3; i--;)
        VecSet(&in, i, VecGet(input, iInput * 3L + i));
    NNEval(that->_nn, (VecFloat*)&in, out);
    for (long i = ISCGetNbClass(that); i--;)
        VecSet(res, iInput * (long)ISCGetNbClass(that) + i,
            VecGet(out, i));
}
// Free memory
VecFree(&out);
// Return the result
return res;
}

// Return the number of int parameters for the criterion 'that'
long ISCRGBGetNbParamInt(const ImgSegmentorCriterionRGB* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    (void)that;
    return 0;
}

// Return the number of float parameters for the criterion 'that'
long ISCRGBGetNbParamFloat(const ImgSegmentorCriterionRGB* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    return NNGetGAAdnFloatLength(that->_nn);
}

// Set the bounds of int parameters for training of the criterion 'that'
void ISCRGBSetBoundsAdnInt(const ImgSegmentorCriterionRGB* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (ga == NULL) {
            PBImgAnalysisErr->_type = PBErTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    // Nothing to do
    (void)that; (void)ga; (void)shift;
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void ISCRGBSetBoundsAdnFloat(const ImgSegmentorCriterionRGB* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (ga == NULL) {
            PBImgAnalysisErr->_type = PBErTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    VecFloat2D bounds = VecFloatCreateStatic2D();
    VecSet(&bounds, 0, -1.0);
    VecSet(&bounds, 1, 1.0);
    for (long iParam = ISCRGBGetNbParamFloat(that); iParam--;) {
        GASetBoundsAdnFloat(ga, iParam + shift, &bounds);
    }
}

// Set the values of int parameters for training of the criterion 'that'
void ISCRGBSetAdnInt(const ImgSegmentorCriterionRGB* const that,
    const GenAlgAdn* const adn, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (adn == NULL) {
            PBImgAnalysisErr->_type = PBErTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
        }
    #endif
}

```

```

        PBErCatch(PBImgAnalysisErr);
    }
#endif
    // Nothing to do
    (void)that; (void)adn; (void)shift;
}

// Set the values of float parameters for training of the criterion
// 'that'
void ISCRGBSetAdnFloat(const ImgSegmentorCriterionRGB* const that,
    const GenAlgAdn* const adn, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (adn == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    const VecFloat* adnF = GAAdnAdnF(adn);
    VecFloat* bases = VecFloatCreate(ISCRGBGetNbParamFloat(that));
    for (int i = ISCRGBGetNbParamFloat(that); i--;)
        VecSet(bases, i, VecGet(adnF, shift + i));
    NNSetBases((NeuraNet*)ISCRGBNeuraNet(that), bases);
    VecFree(&bases);
}

// ---- ImgSegmentorCriterionRGB2HSV

// Create a new ImgSegmentorCriterionRGB2HSV with 'nbClass' output
ImgSegmentorCriterionRGB2HSV* ImgSegmentorCriterionRGB2HSVCreate(
    const int nbClass) {
    #if BUILDMODE == 0
        if (nbClass <= 0) {
            PBImgAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
                nbClass);
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    (void)nbClass;
    // Allocate memory for the new ImgSegmentorCriterionRGB
    ImgSegmentorCriterionRGB2HSV* that = PBErMalloc(PBImgAnalysisErr,
        sizeof(ImgSegmentorCriterionRGB2HSV));
    // Create the parent ImgSegmentorCriterion
    that->_criterion = ImgSegmentorCriterionCreateStatic(nbClass,
        ISCType_RGB2HSV);
    // Return the new ImgSegmentorCriterionRGB
    return that;
}

// Free the memory used by the ImgSegmentorCriterionRGB 'that'
void ImgSegmentorCriterionRGB2HSVFree(
    ImgSegmentorCriterionRGB2HSV** that) {
    if (that == NULL || *that == NULL)
        return;
    // Free memory
    ImgSegmentorCriterionFreeStatic((ImgSegmentorCriterion*)(*that));
}

```



```

    free(*that);
}

// Function which return the JSON encoding of 'that'
void ISCRGB2HSVEncodeAsJSON(
    const ImgSegmentorCriterionRGB2HSV* const that, JSONNode* const json) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    // Nothing to do
    (void)that; (void)json;
}

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionRGB2HSV
bool ISCRGB2HSVDecodeAsJSON(
    ImgSegmentorCriterionRGB2HSV** const that,
    const JSONNode* const json) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
        if (json == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'json' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    // If the criterion exists
    if (*that != NULL) {
        // Free the memory
        ImgSegmentorCriterionRGB2HSVFree(that);
    }
    // Get the number of class
    JSONNode* prop = JSONProperty(json, "_nbClass");
    if (prop == NULL) {
        return false;
    }
    int nbClass = atoi(JSONLabel(JSONValue(prop, 0)));
    // If the number of class is invalid
    if (nbClass < 1)
        // Return the error code
        return false;
    // Create the criterion
    *that = ImgSegmentorCriterionRGB2HSVCreate(nbClass);
    // If we couldn't create the criterion
    if (*that == NULL)
        // Return the failure code
        return false;
    // Return the success code
    return true;
}

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionRGB2HSV that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]

```

```

// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCRGB2HSVPredict(
    const ImgSegmentorCriterionRGB2HSV* const that,
    const VecFloat* input, const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (input == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'input' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (dim == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'dim' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if ((VecGet(dim, 0) * VecGet(dim, 1) * 3) != VecGetDim(input)) {
        PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBImpAnalysisErr->_msg,
            "'input' 's dim is invalid (%ld=%d*d*3)", VecGetDim(input),
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    /*
    printf("ISCRGB2HSVPredict <%.3f,%.3f,%.3f %.3f,%.3f,%.3f ...>\n",
        VecGet(input, 0), VecGet(input, 1), VecGet(input, 2),
        VecGet(input, 3), VecGet(input, 4), VecGet(input, 5));
    */
    (void)that;
    // Calculate the area of the input image
    long area = VecGet(dim, 0) * VecGet(dim, 1);
    // Allocate memory for the result
    VecFloat* res = VecFloatCreate(area * 3L);
    // Loop over the image
    for (long iPos = 0; iPos < area && !PBIA_CtrlC; ++iPos) {
        // Get the pixel
        GBPixel pix = GBColorWhite;
        for (int iRGB = 3; iRGB--;)
            pix._rgba[iRGB] = (unsigned char)round(
                255.0 * VecGet(input, iPos * 3 + iRGB));
        // Convert to HSV
        pix = GBPixelRGB2HSV(&pix);
        // Update the result
        for (int iHSV = 3; iHSV--;)
            VecSet(res, iPos * 3 + iHSV, (float)(pix._hsva[iHSV]) / 255.0);
    }
    // Return the result
    return res;
}

// Return the number of int parameters for the criterion 'that'
long ISCRGB2HSVGetNbParamInt(
    const ImgSegmentorCriterionRGB2HSV* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
    }
#endif
}

```

```

        PBErCatch(PBImgAnalysisErr);
    }
#endif
    (void)that;
    return 0;
}

// Return the number of float parameters for the criterion 'that'
long ISCRGB2HSVGetNbParamFloat(
    const ImgSegmentorCriterionRGB2HSV* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    (void)that;
    return 0;
}

// Set the bounds of int parameters for training of the criterion 'that'
void ISCRGB2HSVSetBoundsAdnInt(
    const ImgSegmentorCriterionRGB2HSV* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (ga == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    // Nothing to do
    (void)that; (void)ga; (void)shift;
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void ISCRGB2HSVSetBoundsAdnFloat(
    const ImgSegmentorCriterionRGB2HSV* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (ga == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    // Nothing to do
    (void)that; (void)ga; (void)shift;
}

```

```

// Set the values of int parameters for training of the criterion 'that'
void ISCRGB2HSVSetAdnInt(const ImgSegmentorCriterionRGB2HSV* const that,
    const GenAlgAdn* const adn, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (adn == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Nothing to do
    (void)that; (void)adn; (void)shift;
}

// Set the values of float parameters for training of the criterion
// 'that'
void ISCRGB2HSVSetAdnFloat(
    const ImgSegmentorCriterionRGB2HSV* const that,
    const GenAlgAdn* const adn, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (adn == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Nothing to do
    (void)that; (void)adn; (void)shift;
}

// ---- ImgSegmentorCriterionDust

// Create a new ImgSegmentorCriterionDust with 'nbClass' output
ImgSegmentorCriterionDust* ImgSegmentorCriterionDustCreate(
    const int nbClass) {
    #if BUILDMODE == 0
        if (nbClass <= 0) {
            PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBIImgAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
                nbClass);
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    (void)nbClass;
    // Allocate memory for the new ImgSegmentorCriterionRGB
    ImgSegmentorCriterionDust* that = PBErrMalloc(PBIImgAnalysisErr,
        sizeof(ImgSegmentorCriterionDust));
    // Create the parent ImgSegmentorCriterion
    that->_criterion = ImgSegmentorCriterionCreateStatic(nbClass,
        ISCType_Dust);
    // Allocate memory for the dust size
    that->_size = VecLongCreate(nbClass);
}

```

```

    // Return the new ImgSegmentorCriterionRGB
    return that;
}

// Free the memory used by the ImgSegmentorCriterionRGB 'that'
void ImgSegmentorCriterionDustFree(
    ImgSegmentorCriterionDust** that) {
    if (that == NULL || *that == NULL)
        return;
    // Free memory
    VecFree(&((*that)->_size));
    ImgSegmentorCriterionFreeStatic((ImgSegmentorCriterion*)(*that));
    free(*that);
}

// Function which return the JSON encoding of 'that'
void ISCDustEncodeAsJSON(
    const ImgSegmentorCriterionDust* const that, JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Encode the dust sizes
    JSONAddProp(json, "_size", VecEncodeAsJSON(that->_size));
}

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionDust
bool ISCDustDecodeAsJSON(
    ImgSegmentorCriterionDust** const that,
    const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (json == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'json' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // If the criterion exists
    if (*that != NULL) {
        // Free the memory
        ImgSegmentorCriterionDustFree(that);
    }
    // Get the number of class
    JSONNode* prop = JSONProperty(json, "_nbClass");
    if (prop == NULL) {
        return false;
    }
    int nbClass = atoi(JSONLabel(JSONValue(prop, 0)));
    // If the number of class is invalid
    if (nbClass < 1)
        // Return the error code
        return false;
    // Create the criterion

```

```

    *that = ImgSegmentorCriterionDustCreate(nbClass);
    // If we couldn't create the criterion
    if (*that == NULL)
        // Return the failure code
        return false;
    // Decode the dust sizes
    prop = JSONProperty(json, "_size");
    if (prop == NULL) {
        return false;
    }
    if (!VecDecodeAsJSON(&((*that)->_size), prop)) {
        return false;
    }
    // Return the success code
    return true;
}

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionDust that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCDustPredict(
    const ImgSegmentorCriterionDust* const that,
    const VecFloat* input, const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (input == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'input' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (dim == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'dim' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if ((VecGet(dim, 0) * VecGet(dim, 1)) != VecGetDim(input)) {
        PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBIImgAnalysisErr->_msg,
            "'input' 's dim is invalid (%ld=%d*d)", VecGetDim(input),
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    /*
    printf("ISCDustPredict <%.3f,%.3f,%.3f %.3f,%.3f,%.3f ...>\n",
        VecGet(input, 0), VecGet(input, 1), VecGet(input, 2),
        VecGet(input, 3), VecGet(input, 4), VecGet(input, 5));
    */
    (void)that; (void)input;
    // Calculate the area of the input image
    long area = VecGet(dim, 0) * VecGet(dim, 1);
    // Allocate memory for the result
    VecFloat* res = VecFloatCreate(area * 3L);
    // Loop over the image
    for (long iPos = 0; iPos < area; ++iPos) {

```

```

        // Get the pixel
        GBPixel pix = GBColorWhite;
        for (int iRGB = 3; iRGB--;)
            pix._rgba[iRGB] = (unsigned char)round(
                255.0 * VecGet(input, iPos * 3 + iRGB));
        // Convert to HSV
        pix = GBPixelDust(&pix);
        // Update the result
        for (int iHSV = 3; iHSV--;)
            VecSet(res, iPos * 3 + iHSV, (float)(pix._hsva[iHSV]) / 255.0);
    */

    }
    // Return the result
    return res;
}

// Return the number of int parameters for the criterion 'that'
long ISCDustGetNbParamInt(
    const ImgSegmentorCriterionDust* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    return ISCGetNbClass(that);
}

// Return the number of float parameters for the criterion 'that'
long ISCDustGetNbParamFloat(
    const ImgSegmentorCriterionDust* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    (void)that;
    return 0;
}

// Set the bounds of int parameters for training of the criterion 'that'
void ISCDustSetBoundsAdnInt(
    const ImgSegmentorCriterionDust* const that,
    GenAlg* const ga, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
        if (ga == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'ga' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    VecLong2D bounds = VecLongCreateStatic2D();
    VecSet(&bounds, 0, 0);
}

```

```

    VecSet(&bounds, 1, 100);
    for (long iParam = ISCDustGetNbParamInt(that); iParam--;) {
        GASetBoundsAdnInt(ga, iParam + shift, &bounds);
    }
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void ISCDustSetBoundsAdnFloat(
    const ImgSegmentorCriterionDust* const that,
    GenAlg* const ga, const long shift) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (ga == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Nothing to do
    (void)that; (void)ga; (void)shift;
}

// Set the values of int parameters for training of the criterion 'that'
void ISCDustSetAdnInt(const ImgSegmentorCriterionDust* const that,
    const GenAlgAdn* const adn, const long shift) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (adn == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    const VecLong* adnI = GAAdnAdnI(adn);
    for (int i = ISCDustGetNbParamInt(that); i--;)
        ISCDustSetSize(that, i, VecGet(adnI, shift + i));
}

// Set the values of float parameters for training of the criterion
// 'that'
void ISCDustSetAdnFloat(
    const ImgSegmentorCriterionDust* const that,
    const GenAlgAdn* const adn, const long shift) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (adn == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
}

```



```

    }
#endif
    // Nothing to do
    (void)that; (void)adn; (void)shift;
}

// ---- ImgSegmentorCriterionTex

// Create a new ImgSegmentorCriterionTex with 'nbClass' output,
// 'rank' hidden layers and 3^'size' x 3^'size' down to 1x1 square
// fragments of the image as input
ImgSegmentorCriterionTex* ImgSegmentorCriterionTexCreate(
    const int nbClass, const int rank, const int size) {
    #if BUILDMODE == 0
        if (nbClass <= 0) {
            PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBImpAnalysisErr->_msg, "'nbClass' is invalid (%d>0)",
                nbClass);
            PBErrCatch(PBImpAnalysisErr);
        }
        if (rank <= 0) {
            PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBImpAnalysisErr->_msg, "'rank' is invalid (%d>0)",
                rank);
            PBErrCatch(PBImpAnalysisErr);
        }
        if (size <= 0) {
            PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBImpAnalysisErr->_msg, "'size' is invalid (%d>0)",
                size);
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    // Allocate memory for the new ImgSegmentorCriterionTex
    ImgSegmentorCriterionTex* that = PBErrMalloc(PBImpAnalysisErr,
        sizeof(ImgSegmentorCriterionTex));
    // Create the parent ImgSegmentorCriterion
    that->_criterion = ImgSegmentorCriterionCreateStatic(nbClass,
        ISCType_Tex);
    // Set the properties
    that->_size = size;
    that->_rank = rank;
    // Create the NeuralNet
    const int nbInput = 3 * (1 + (size == 1 ? 0 : (size - 1) * 9));
    const int nbHiddenPerLayer = nbInput * nbClass;
    VecLong* hidden = VecLongCreate(rank);
    for (int iLayer = rank; iLayer--;)
        VecSet(hidden, iLayer, nbHiddenPerLayer);
    that->_nn = NeuralNetCreateFullyConnected(nbInput, nbClass, hidden);
    VecFree(&hidden);
    // Return the new ImgSegmentorCriterionTex
    return that;
}

// Free the memory used by the ImgSegmentorCriterionTex 'that'
void ImgSegmentorCriterionTexFree(ImgSegmentorCriterionTex** that) {
    if (that == NULL || *that == NULL)
        return;
    // Free memory
    ImgSegmentorCriterionFreeStatic((ImgSegmentorCriterion*)(*that));
    NeuralNetFree(&((*that)->_nn));
    free(*that);
}

```

```

}

// Function which return the JSON encoding of 'that'
void ISCTexEncodeAsJSON(
    const ImgSegmentorCriterionTex* const that, JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Declare a buffer to convert value into string
    char val[100];
    // Rank
    sprintf(val, "%d", ISCTexGetRank(that));
    JSONAddProp(json, "_rank", val);
    // Size
    sprintf(val, "%d", ISCTexGetSize(that));
    JSONAddProp(json, "_size", val);
    // NeuraNet model
    JSONAddProp(json, "_neuranet", NNEncodeAsJSON(that->_nn));
}

// Function which decodes the JSON encoding of a
// ImgSegmentorCriterionTex
bool ISCTexDecodeAsJSON(
    ImgSegmentorCriterionTex** const that, const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (json == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'json' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // If the criterion exists
    if (*that != NULL) {
        // Free the memory
        ImgSegmentorCriterionTexFree(that);
    }
    // Get the number of class
    JSONNode* prop = JSONProperty(json, "_nbClass");
    if (prop == NULL) {
        return false;
    }
    int nbClass = atoi(JSONLabel(JSONValue(prop, 0)));
    // If the number of class is invalid
    if (nbClass < 1)
        // Return the error code
        return false;
    // Get the size
    prop = JSONProperty(json, "_size");
    if (prop == NULL) {
        return false;
    }
    int size = atoi(JSONLabel(JSONValue(prop, 0)));
    // If the size is invalid

```

```

    if (size < 1)
        // Return the error code
        return false;
    // Get the rank
    prop = JSONProperty(json, "_rank");
    if (prop == NULL) {
        return false;
    }
    int rank = atoi(JSONLabel(JSONValue(prop, 0)));
    // If the rank is invalid
    if (rank < 1)
        // Return the error code
        return false;
    // Create the criterion
    *that = ImgSegmentorCriterionTexCreate(nbClass, rank, size);
    // If we couldn't create the criterion
    if (*that == NULL)
        // Return the failure code
        return false;
    // Decode the NeuraNet
    prop = JSONProperty(json, "_neuranet");
    if (prop == NULL) {
        return false;
    }
    if (!NNDecodeAsJSON(&((*that)->_nn), prop))
        return false;
    // Return the success code
    return true;
}

// Make the prediction on the 'input' values with the
// ImgSegmentorCriterionTex that
// 'input' 's format is 3*width*height, values in [0.0, 1.0]
// Return values are nbClass*width*height, values in [-1.0, 1.0]
VecFloat* ISCTexPredict(const ImgSegmentorCriterionTex* const that,
    const VecFloat* input, const VecShort2D* const dim) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (input == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'input' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (dim == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'dim' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
    if ((VecGet(dim, 0) * VecGet(dim, 1) * 3) != VecGetDim(input)) {
        PBImpAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBImpAnalysisErr->_msg,
            "'input' 's dim is invalid (%ld=%d*d*3)", VecGetDim(input),
            VecGet(dim, 0), VecGet(dim, 1));
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    //printf("dim ");VecPrint(dim,stdout);printf("\n");
    // Calculate the area of the input image

```

```

    long area = VecGet(dim, 0) * VecGet(dim, 1);
    //printf("area %ld\n",area);
    // Allocate memory for the result
    VecFloat* res = VecFloatCreate(area * (long)ISCGetNbClass(that));
    // Declare variables to memorize the input/output of the NeuralNet
    VecFloat* out = VecFloatCreate(ISCGetNbClass(that));
    int nbIn = 3 * (1 + (ISCTexGetSize(that) == 1 ? 0 :
        (ISCTexGetSize(that) - 1) * 9));
    //printf("nbIn %d\n",nbIn);
    VecFloat* in = VecFloatCreate(nbIn);
    // Calculate the size of the biggest fragment
    int sizeFragMax = powi(3, ISCTexGetSize(that) - 1);
    //printf("sizeFragMax %d\n", sizeFragMax);
    // Declare a variable to memorize the index of current pixel in the
    // input
    long iInput = 0;
    // Declare a variable to memorize the dimension of the fragment
    VecShort2D dimFrag = VecShortCreateStatic2D();
    // Loop on the image
    VecShort2D pos = VecShortCreateStatic2D();
    do {
        // Ignore the border of the image where there is not enough
        // space to create the fragments
        if (VecGet(&pos, 0) >= sizeFragMax - 1 &&
            VecGet(&pos, 0) <= (VecGet(dim, 0) - sizeFragMax) &&
            VecGet(&pos, 1) >= sizeFragMax - 1 &&
            VecGet(&pos, 1) <= (VecGet(dim, 1) - sizeFragMax)) {
            //printf("iInput %ld\n",iInput);
            //printf("pos ");VecPrint(&pos,stdout);printf("\n");
            // Current pixel (fragment of size 1x1)
            for (long i = 3; i--;)
                VecSet(in, i, VecGet(input, iInput * 3L + i));
            // Loop on fragment sizes bigger than 1x1
            for (int iSize = 1; iSize < ISCTexGetSize(that); ++iSize) {
            //printf("iSize %d\n",iSize);
                // Get the size of the current fragment
                int sizeFrag = powi(3, iSize);
            //printf("sizeFrag %d\n",sizeFrag);
                VecSet(&dimFrag, 0, sizeFrag);
                VecSet(&dimFrag, 1, sizeFrag);
                // Get the area of the frag
                long areaFrag = sizeFrag * sizeFrag;
            //printf("areaFrag %ld\n",areaFrag);
                // Get the half size of the current fragment
                int halfSizeFrag = (sizeFrag - 1) / 2;
            //printf("halfSizeFrag %d\n",halfSizeFrag);
                // Create the matrix of fragments' start position relative to
                // the current pixel
                int relPos[18] = {
                    sizeFrag - 1, sizeFrag - 1,
                    sizeFrag - 1, halfSizeFrag,
                    sizeFrag - 1, 0,
                    halfSizeFrag, sizeFrag - 1,
                    halfSizeFrag, halfSizeFrag,
                    halfSizeFrag, 0,
                    0, sizeFrag - 1,
                    0, halfSizeFrag,
                    0, 0
                };
                // Loop on the 9 fragments for the current size
                for (int iFrag = 9; iFrag--;) {
            //printf("iFrag %d\n", iFrag);

```

```

        // Declare a variable to memorize the average value
        float avg[3] = {0.0, 0.0, 0.0};
        // Get the starting and ending pos for this fragment
        VecShort2D startPosFrag = VecShortCreateStatic2D();
        VecSet(&startPosFrag, 0,
            VecGet(&pos, 0) - relPos[iFrag * 2]);
        VecSet(&startPosFrag, 1,
            VecGet(&pos, 1) - relPos[iFrag * 2 + 1]);
        VecShort2D endPosFrag = VecShortCreateStatic2D();
        VecSet(&endPosFrag, 0, VecGet(&startPosFrag, 0) + sizeFrag);
        VecSet(&endPosFrag, 1, VecGet(&startPosFrag, 1) + sizeFrag);
        //printf("startPosFrag ");VecPrint(&startPosFrag,stdout);printf("\n");
        //printf("endPosFrag ");VecPrint(&endPosFrag,stdout);printf("\n");
        // Loop on the fragment to calculate the average rgb value
        VecShort2D posFrag = startPosFrag;
        do {
        //printf("posFrag ");VecPrint(&posFrag,stdout);printf("\n");
            long iPosFrag = GBPosIndex(&posFrag, dim) * 3;
        //printf("iPosFrag %ld\n",iPosFrag);
            for (long i = 3; i--;)
                avg[i] += VecGet(input, iPosFrag + i);
            while (VecShiftStep(&posFrag, &startPosFrag, &endPosFrag));
            for (long i = 3; i--;)
                avg[i] /= (float)areaFrag;
        //printf("avg %f %f %f\n",avg[0],avg[1],avg[2]);
            // Set the average value in the input vector
        //printf("shift set in %d\n", 3 * (1 + (iSize - 1) * 9 + iFrag));
            for (long i = 3; i--;)
                VecSet(in, 3 * (1 + (iSize - 1) * 9 + iFrag) + i, avg[i]);
        //printf("in ");VecPrint(in,stdout);printf("\n");
        }
    }
    // Apply the NeuraNet on inputs
    NNEval(that->_nn, in, out);
    //printf("out ");VecPrint(out,stdout);printf("\n");
    // Store the result
    for (long i = ISCGetNbClass(that); i--;)
        VecSet(res, iInput * (long)ISCGetNbClass(that) + i,
            VecGet(out, i));
    }
    // Increment the index of the current pixel in input
    ++iInput;
    } while (VecStep(&pos, dim) && !PBIA_CtrlC);
    // Free memory
    VecFree(&out);
    // Return the result
    return res;
}

// Return the number of int parameters for the criterion 'that'
long ISCTexGetNbParamInt(const ImgSegmentorCriterionTex* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    (void)that;
    return 0;
}

```

```

// Return the number of float parameters for the criterion 'that'
long ISCTexGetNbParamFloat(const ImgSegmentorCriterionTex* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return NNGetGAAdnFloatLength(that->_nn);
}

// Set the bounds of int parameters for training of the criterion 'that'
void ISCTexSetBoundsAdnInt(const ImgSegmentorCriterionTex* const that,
    GenAlg* const ga, const long shift) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (ga == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Nothing to do
    (void)that; (void)ga; (void)shift;
}

// Set the bounds of float parameters for training of the criterion
// 'that'
void ISCTexSetBoundsAdnFloat(const ImgSegmentorCriterionTex* const that,
    GenAlg* const ga, const long shift) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (ga == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'ga' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    VecFloat2D bounds = VecFloatCreateStatic2D();
    VecSet(&bounds, 0, -1.0);
    VecSet(&bounds, 1, 1.0);
    for (long iParam = ISCTexGetNbParamFloat(that); iParam--;) {
        GASetBoundsAdnFloat(ga, iParam + shift, &bounds);
    }
}

// Set the values of int parameters for training of the criterion 'that'
void ISCTexSetAdnInt(const ImgSegmentorCriterionTex* const that,
    const GenAlgAdn* const adn, const long shift) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
    }
#endif
}

```

```

        PBErCatch(PBImgAnalysisErr);
    }
    if (adn == NULL) {
        PBImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    // Nothing to do
    (void)that; (void)adn; (void)shift;
}

// Set the values of float parameters for training of the criterion
// 'that'
void ISCTexSetAdnFloat(const ImgSegmentorCriterionTex* const that,
    const GenAlgAdn* const adn, const long shift) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (adn == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'ga' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
    const VecFloat* adnF = GAAAdnAdnF(adn);
    VecFloat* bases = VecFloatCreate(ISCTexGetNbParamFloat(that));
    for (int i = ISCTexGetNbParamFloat(that); i--;)
        VecSet(bases, i, VecGet(adnF, shift + i));
    NNSetBases((NeuraNet*)ISCTexNeuraNet(that), bases);
    VecFree(&bases);
}

// ----- General functions -----

// ===== Functions implementation =====

// Return the Jaccard index (aka intersection over union) of the
// image 'that' and 'tho' for pixels of color 'rgba'
// 'that' and 'tho' must have same dimensions
float IntersectionOverUnion(const GenBrush* const that,
    const GenBrush* const tho, const GBPixel* const rgba) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'that' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (tho == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'tho' is null");
            PBErCatch(PBImgAnalysisErr);
        }
        if (rgba == NULL) {
            PBImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImgAnalysisErr->_msg, "'rgba' is null");
            PBErCatch(PBImgAnalysisErr);
        }
    #endif
}

```

```

    if (!VecIsEqual(GBDim(that), GBDim(tho))) {
        PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBIImgAnalysisErr->_msg,
            "'that' and 'tho' have different dimensions");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    // Declare two variables to count the number of pixels in
    // intersection and union
    long nbUnion = 0;
    long nbInter = 0;
    // Declare a variable to loop through pixels
    VecShort2D pos = VecShortCreateStatic2D();
    // Loop through pixels
    do {
        // If the pixel is in the intersection
        if (GBPixelIsSame(GBFinalPixel(that, &pos), rgba) &&
            GBPixelIsSame(GBFinalPixel(tho, &pos), rgba)) {
            // Increment the number of pixels in intersection
            ++nbInter;
        }
        // If the pixel is in the union
        if (GBPixelIsSame(GBFinalPixel(that, &pos), rgba) ||
            GBPixelIsSame(GBFinalPixel(tho, &pos), rgba)) {
            // Increment the number of pixels in union
            ++nbUnion;
        }
    } while (VecStep(&pos, GBDim(that)));
    // Calculate the intersection over union
    float iou = (float)nbInter / (float)nbUnion;
    // Return the result
    return iou;
}

// Return the similarity coefficient of the images 'that' and 'tho'
// (i.e. the sum of the distances of pixels at the same position
// over the whole image)
// Return a value in [0.0, 1.0], 1.0 means the two images are
// identical, 0.0 means they are binary black and white with each
// pixel in one image the opposite of the corresponding pixel in the
// other image.
// 'that' and 'tho' must have same dimensions
float GBSimilarityCoeff(const GenBrush* const that,
    const GenBrush* const tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (tho == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'tho' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
        if (!VecIsEqual(GBDim(that), GBDim(tho))) {
            PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
            sprintf(PBIImgAnalysisErr->_msg,
                "'that' and 'tho' have different dimensions");
            PBErrCatch(PBIImgAnalysisErr);
        }
    }
#endif
}

```



```

// Declare a variable to calculate the result
float res = 0.0;
// Declare a variable to loop through pixels
VecShort2D pos = VecShortCreateStatic2D();
// Loop through pixels
do {
    const GBPixel* pixA = GBFinalPixel(that, &pos);
    const GBPixel* pixB = GBFinalPixel(tho, &pos);
    res += sqrt(
        fsquare((float)(pixA->_rgba[0]) - (float)(pixB->_rgba[0])) +
        fsquare((float)(pixA->_rgba[1]) - (float)(pixB->_rgba[1])) +
        fsquare((float)(pixA->_rgba[2]) - (float)(pixB->_rgba[2])) +
        fsquare((float)(pixA->_rgba[3]) - (float)(pixB->_rgba[3])));
} while (VecStep(&pos, GBDim(that)));
// Calculate the result
res /= (float)GBArea(that) * 510.0;
// Return the result
return 1.0 - res;
}

```

2.2 pbimganalysis-inline.c

```

// ===== PBIMGANALYSIS_INLINE.C =====

// ===== Functions implementation =====

// Get the GenBrush of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const GenBrush* IKMCImg(const ImgKMeansClusters* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    return that->_img;
}

// Set the GenBrush of the ImgKMeansClusters 'that' to 'img'
#if BUILDMODE != 0
inline
#endif
void IKMCSetImg(ImgKMeansClusters* const that,
    const GenBrush* const img) {
    #if BUILDMODE == 0
        if (that == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
        if (img == NULL) {
            PBImpAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBImpAnalysisErr->_msg, "'img' is null");
            PBErrCatch(PBImpAnalysisErr);
        }
    #endif
    that->_img = img;
}

```

```

}

// Get the KMeansClusters of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
const KMeansClusters* IKMCKMeansClusters(
    const ImgKMeansClusters* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return &(that->_kmeansClusters);
}

// Set the size of the cells of the ImgKMeansClusters 'that' to
// 2*'size'+1
#if BUILDMODE != 0
inline
#endif
void IKMCSetSizeCell(ImgKMeansClusters* const that, const int size) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (size < 0) {
        PBIImgAnalysisErr->_type = PBErrTypeInvalidArg;
        sprintf(PBIImgAnalysisErr->_msg, "'size' is invalid (%d>=0)", size);
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    that->_size = size;
}

// Get the size of the cells of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int IKMCGetSizeCell(const ImgKMeansClusters* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return 2 * that->_size + 1;
}

// Get the number of cluster of the ImgKMeansClusters 'that'
#if BUILDMODE != 0
inline
#endif
int IKMCGetK(const ImgKMeansClusters* const that) {
#if BUILDMODE == 0
    if (that == NULL) {

```

```

        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return KMeansClustersGetK(&(that->_kmeansClusters));
}

// Return the nb of criterion of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
long ISGetNbCriterion(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return GenTreeGetSize(ISCriteria(that));
}

// Return the nb of classes of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetNbClass(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_nbClass;
}

// Return the nb of criterion of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
const GenTree* ISCriteria(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return &(that->_criteria);
}

// Add a new ImageSegmentorCriterionRGB to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
// Return the added criterion if successful, null else
#if BUILDMODE != 0
inline
#endif
ImgSegmentorCriterionRGB* ISAddCriterionRGB(ImgSegmentor* const that,
void* const parent) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
// Create and add the criterion to the set of criteria
if (parent == NULL) {
    ImgSegmentorCriterionRGB* criterion =
        ImgSegmentorCriterionRGBCreate(ISGetNbClass(that));
    GenTreeAppendData(&(that->_criteria), criterion);
    return criterion;
} else {
    GenTreeIterDepth iter =
        GenTreeIterDepthCreateStatic(&(that->_criteria));
    ImgSegmentorCriterionRGB* criterion =
        ImgSegmentorCriterionRGBCreate(ISGetNbClass(that));
    bool ret = GenTreeAppendToNode(
        &(that->_criteria), criterion, parent, &iter);
    GenTreeIterFreeStatic(&iter);
    if (ret) {
        return criterion;
    } else {
        ImgSegmentorCriterionRGBFree(&criterion);
        return NULL;
    }
}
return NULL;
}

// Add a new ImageSegmentorCriterionTex to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
// Return the added criterion if successful, null else
#if BUILDMODE != 0
inline
#endif
ImgSegmentorCriterionTex* ISAddCriterionTex(ImgSegmentor* const that,
    void* const parent, const int rank, const int size) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
// Create and add the criterion to the set of criteria
if (parent == NULL) {
    ImgSegmentorCriterionTex* criterion =
        ImgSegmentorCriterionTexCreate(ISGetNbClass(that), rank, size);
    GenTreeAppendData(&(that->_criteria), criterion);
    return criterion;
} else {
    GenTreeIterDepth iter =
        GenTreeIterDepthCreateStatic(&(that->_criteria));
    ImgSegmentorCriterionTex* criterion =
        ImgSegmentorCriterionTexCreate(ISGetNbClass(that), rank, size);
    bool ret = GenTreeAppendToNode(
        &(that->_criteria), criterion, parent, &iter);
    GenTreeIterFreeStatic(&iter);
    if (ret) {

```

```

        return criterion;
    } else {
        ImgSegmentorCriterionTexFree(&criterion);
        return NULL;
    }
}
return NULL;
}

// Add a new ImageSegmentorCriterionRGB2HSV to the ImgSegmentor 'that'
// under the node 'parent'
// If 'parent' is null it is inserted to the root of the ImgSegmentor
// Return the added criterion if successful, null else
#if BUILDMODE != 0
inline
#endif
ImgSegmentorCriterionRGB2HSV* ISAddCriterionRGB2HSV(
    ImgSegmentor* const that, void* const parent) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
    // Create and add the criterion to the set of criteria
    if (parent == NULL) {
        ImgSegmentorCriterionRGB2HSV* criterion =
            ImgSegmentorCriterionRGB2HSVCreate(ISGetNbClass(that));
        GenTreeAppendData(&(that->_criteria), criterion);
        return criterion;
    } else {
        GenTreeIterDepth iter =
            GenTreeIterDepthCreateStatic(&(that->_criteria));
        ImgSegmentorCriterionRGB2HSV* criterion =
            ImgSegmentorCriterionRGB2HSVCreate(ISGetNbClass(that));
        bool ret = GenTreeAppendToNode(
            &(that->_criteria), criterion, parent, &iter);
        GenTreeIterFreeStatic(&iter);
        if (ret) {
            return criterion;
        } else {
            ImgSegmentorCriterionRGB2HSVFree(&criterion);
            return NULL;
        }
    }
    return NULL;
}

// Return the flag controlling the binarization of the result of
// prediction of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
bool ISGetFlagBinaryResult(const ImgSegmentor* const that) {
    if BUILDMODE == 0
        if (that == NULL) {
            PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
            sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
            PBErrCatch(PBIImgAnalysisErr);
        }
    #endif
}

```

```

    return that->_flagBinaryResult;
}

// Return the threshold controlling the binarization of the result of
// prediction of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
float ISGetThresholdBinaryResult(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_thresholdBinaryResult;
}

// Set the flag controlling the binarization of the result of
// prediction of the ImgSegmentor 'that' to 'flag'
#if BUILDMODE != 0
inline
#endif
void ISSetFlagBinaryResult(ImgSegmentor* const that,
    const bool flag) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    that->_flagBinaryResult = flag;
}

// Set the threshold controlling the binarization of the result of
// prediction of the ImgSegmentor 'that' to 'threshold'
#if BUILDMODE != 0
inline
#endif
void ISSetThresholdBinaryResult(ImgSegmentor* const that,
    const float threshold) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    that->_thresholdBinaryResult = threshold;
}

// Return the number of epoch for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
unsigned int ISGetNbEpoch(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
    }
#endif
}

```

```

        PBErCatch(PBImgAnalysisErr);
    }
#endif
    return that->_nbEpoch;
}

// Set the number of epoch for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetNbEpoch(ImgSegmentor* const that, unsigned int nb) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    that->_nbEpoch = nb;
}

// Return the nb of class of the ImgSegmentorCriterion 'that'
#if BUILDMODE != 0
inline
#endif
int _ISGetNbClass(const ImgSegmentorCriterion* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    return that->_nbClass;
}

// Return the size of the pool for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizePool(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
#endif
    return that->_sizePool;
}

// Set the size of the pool for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizePool(ImgSegmentor* const that, int nb) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImgAnalysisErr->_type = PBErTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErCatch(PBImgAnalysisErr);
    }
}

```

```

#endif
    that->_sizePool = nb;
}

// Return the nb of elites for training the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetNbElite(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_nbElite;
}

// Set the nb of elites for training the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetNbElite(ImgSegmentor* const that, int nb) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    that->_nbElite = nb;
}

// Return the threshold controlling the stop of the training
#if BUILDMODE != 0
inline
#endif
float ISGetTargetBestValue(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_targetBestValue;
}

// Set the threshold controlling the stop of the training to 'val'
// Clip the value to [0.0, 1.0]
#if BUILDMODE != 0
inline
#endif
void ISSetTargetBestValue(ImgSegmentor* const that, const float val) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
}

```



```

    that->_targetBestValue = MIN(1.0, MAX(0.0, val));
}

// Return the flag for the TextOMeter of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
bool ISGetFlagTextOMeter(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    return that->_flagTextOMeter;
}

// Return the max nb of adns of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizeMaxPool(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    return that->_sizeMaxPool;
}

// Return the min nb of adns of the ImgSegmentor 'that'
#if BUILDMODE != 0
inline
#endif
int ISGetSizeMinPool(const ImgSegmentor* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    return that->_sizeMinPool;
}

// Set the min nb of adns of the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizeMaxPool(ImgSegmentor* const that, const int nb) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    that->_sizeMaxPool = MAX(ISGetSizeMinPool(that), nb);
}

```

```

// Set the min nb of adns of the ImgSegmentor 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void ISSetSizeMinPool(ImgSegmentor* const that, const int nb) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    that->_sizeMinPool = MIN(ISGetSizeMaxPool(that), nb);
}

// ---- ImgSegmentorCriterionRGB

// Return the NeuraNet of the ImgSegmentorCriterionRGB 'that'
#if BUILDMODE != 0
inline
#endif
const NeuraNet* ISCRGBNeuraNet(
    const ImgSegmentorCriterionRGB* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    return that->_nn;
}

// ---- ImgSegmentorCriterionDust

// Return the dust size of the ImgSegmentorCriterionDust 'that' for
// the class 'iClass'
#if BUILDMODE != 0
inline
#endif
long ISCDustSize(
    const ImgSegmentorCriterionDust* const that, const int iClass) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBImpAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImpAnalysisErr);
    }
#endif
    return VecGet(that->_size, iClass);
}

// Set the dust size of the ImgSegmentorCriterionDust 'that' for
// the class 'iClass' to 'size'
#if BUILDMODE != 0
inline
#endif
void ISCDustSetSize(
    const ImgSegmentorCriterionDust* const that, const int iClass,
    const long size) {
#if BUILDMODE == 0

```

```

    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    VecSet(that->_size, iClass, size);
}

// ---- ImgSegmentorCriterionTex

// Return the NeuraNet of the ImgSegmentorCriterionTex 'that'
#if BUILDMODE != 0
inline
#endif
const NeuraNet* ISCTexNeuraNet(
    const ImgSegmentorCriterionTex* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_nn;
}

// Return the rank of the ImgSegmentorCriterionTex 'that'
#if BUILDMODE != 0
inline
#endif
int ISCTexGetRank(const ImgSegmentorCriterionTex* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_rank;
}

// Return the size of the ImgSegmentorCriterionTex 'that'
#if BUILDMODE != 0
inline
#endif
int ISCTexGetSize(const ImgSegmentorCriterionTex* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeNullPointer;
        sprintf(PBIImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBIImgAnalysisErr);
    }
#endif
    return that->_size;
}

```

3 Makefile

```
# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: pbmake_wget main

# Automatic installation of the repository PBMake in the parent folder
pbmake_wget:
if [ ! -d ../PBMake ]; then wget https://github.com/BayashiPascal/PBMake/archive/master.zip; unzip master.zip; rm -f

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Rules to make the executable
repo=pbimganalysis
$(repo)_EXENAME: \
$(repo)_EXENAME.o \
$(repo)_EXE_DEP \
$(repo)_DEP
$(COMPILER) 'echo "$(repo)_EXE_DEP" "$(repo)_EXENAME.o" | tr ' ' '\n' | sort -u' $(LINK_ARG) $(repo)_LINK_ARG

$(repo)_EXENAME.o: \
$(repo)_DIR/$$(repo)_EXENAME.c \
$(repo)_INC_H_EXE \
$(repo)_EXE_DEP
$(COMPILER) $(BUILD_ARG) $(repo)_BUILD_ARG 'echo "$(repo)_INC_DIR" | tr ' ' '\n' | sort -u' -c $(repo)_DIR/
```

4 Unit tests

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <math.h>
#include "pbimganalysis.h"

void UnitTestImgKMeansClusters() {
    srandom(1);
    for (int size = 0; size < 2; ++size) {
        for (int K = 2; K <= 3; ++K) {
            char* fileName = "ImgKMeansClustersTest/imgkmeanscluster.tga";
            GenBrush* img = GBCreateFromFile(fileName);
            ImgKMeansClusters clusters = ImgKMeansClustersCreateStatic(
                img, KMeansClustersSeed_Forgy, size);
            IKMCSave(&clusters, K);

            FILE* fd = fopen("./imgkmeanscluster.txt", "w");
            if (!IKMCSave(&clusters, fd, false)) {
                PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
                sprintf(PBImgAnalysisErr->_msg, "IKMCSave NOK");
                PBErrCatch(PBImgAnalysisErr);
            }
            fclose(fd);
        }
    }
}
```

```

    fd = fopen("./imgkmeanscluster.txt", "r");
    if (!IKMCLoad(&clusters, fd)) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg, "IKMCLoad NOK");
        PBErrCatch(PBIImgAnalysisErr);
    }
    IKMCSetImg(&clusters, img);
    fclose(fd);

    printf("%s size K=%d cell=%d:\n",
        fileName, K, IKMCGetSizeCell(&clusters));
    IKMCPrintln(&clusters, stdout);
    IKMCCLuster(&clusters);
    char fileNameOut[50] = {'\0'};
    sprintf(fileNameOut,
        ". /ImgKMeansClustersTest/imgkmeanscluster%02d-%02d.tga", K, size);
    GBSetFileName(img, fileNameOut);
    GBRender(img);
    GBFree(&img);
    ImgKMeansClustersFreeStatic(&clusters);
}
}
printf("UnitTestImgKMeansClusters OK\n");
}

void UnitTestIntersectionOverUnion() {
    char* fileNameA = "./iou1.tga";
    GenBrush* imgA = GBCreateFromFile(fileNameA);
    char* fileNameB = "./iou2.tga";
    GenBrush* imgB = GBCreateFromFile(fileNameB);
    GBPixel rgba = GBColorBlack;
    float iou = IntersectionOverUnion(imgA, imgB, &rgba);
    if (!ISEQUALF(iou, 6.0 / 10.0)) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg, "IntersectionOverUnion failed");
        PBErrCatch(PBIImgAnalysisErr);
    }
    GBFree(&imgA);
    GBFree(&imgB);
    printf("UnitTestIntersectionOverUnion OK\n");
}

void UnitTestGBSimilarityCoefficient() {
    char* fileNameA = "./iou1.tga";
    GenBrush* imgA = GBCreateFromFile(fileNameA);
    char* fileNameB = "./iou2.tga";
    GenBrush* imgB = GBCreateFromFile(fileNameB);
    float sim = GBSimilarityCoeff(imgA, imgA);
    if (!ISEQUALF(sim, 1.0)) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg, "GBSimilarityCoefficient failed");
        PBErrCatch(PBIImgAnalysisErr);
    }
    sim = GBSimilarityCoeff(imgA, imgB);
    if (!ISEQUALF(sim, 0.965359)) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg, "GBSimilarityCoefficient failed");
        PBErrCatch(PBIImgAnalysisErr);
    }
    GBFree(&imgA);
    GBFree(&imgB);
    printf("UnitTestIntersectionOverUnion OK\n");
}

```

```

}

void UnitTestImgSegmentorRGB() {
    int nbClass = 2;
    ImgSegmentorCriterionRGB* criterion =
        ImgSegmentorCriterionRGBCreate(nbClass);
    if (ISGetNbClass(criterion) != nbClass) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg,
            "ImgSegmentorCriterionRGBCreate failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    int imgArea = 4;
    VecFloat* input = VecFloatCreate(imgArea * 3);
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2);
    VecSet(&dim, 1, 2);
    VecFloat* output = ISCRGBPredict(criterion, input, &dim);
    if (VecGetDim(output) != imgArea * nbClass) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "ISCRGBPredict failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    VecFree(&input);
    VecFree(&output);
    ImgSegmentorCriterionRGBFree(&criterion);
    printf("UnitTestImgSegmentorRGB OK\n");
}

void UnitTestImgSegmentorCreateFree() {
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    if (segmentor._nbClass != nbClass ||
        segmentor._flagBinaryResult != false ||
        segmentor._nbEpoch != 1 ||
        segmentor._flagTextOMeter != false ||
        segmentor._textOMeter != NULL ||
        !ISEQUALF(segmentor._thresholdBinaryResult, 0.5) ||
        !ISEQUALF(segmentor._targetBestValue, 0.9999)) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "ImgSegmentorCreateStatic failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    ImgSegmentorFreeStatic(&segmentor);
    printf("UnitTestImgSegmentorCreateFree OK\n");
}

void UnitTestImgSegmentorAddCriterionGetSet() {
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    if (ISCriteria(&segmentor) != &(segmentor._criteria)) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "ISCriteria failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (ISGetNbClass(&segmentor) != nbClass) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "ISGetNbClass failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (ISGetFlagTextOMeter(&segmentor) != false) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

    sprintf(PBImgAnalysisErr->_msg, "ISGetFlagTextOMeter failed");
    PBErCatch(PBImgAnalysisErr);
}
ISSetFlagTextOMeter(&segmentor, true);
if (ISGetFlagTextOMeter(&segmentor) != true) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISSetFlagTextOMeter failed");
    PBErCatch(PBImgAnalysisErr);
}
if (ISGetNbCriterion(&segmentor) != 0) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetNbCriterion failed");
    PBErCatch(PBImgAnalysisErr);
}
if (ISAddCriterionRGB(&segmentor, NULL) == NULL ||
    GenTreeGetSize(ISCriteria(&segmentor)) != 1) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISAddCriterion failed");
    PBErCatch(PBImgAnalysisErr);
}
if (ISGetNbCriterion(&segmentor) != 1) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetNbCriterion failed");
    PBErCatch(PBImgAnalysisErr);
}
ISSetFlagBinaryResult(&segmentor, true);
if (segmentor._flagBinaryResult != true) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISSetFlagBinaryResult failed");
    PBErCatch(PBImgAnalysisErr);
}
if (ISGetFlagBinaryResult(&segmentor) != true) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetFlagBinaryResult failed");
    PBErCatch(PBImgAnalysisErr);
}
ISSetThresholdBinaryResult(&segmentor, 1.0);
if (!ISEQUALF(segmentor._thresholdBinaryResult, 1.0)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISSetThrehsoldBinaryResult failed");
    PBErCatch(PBImgAnalysisErr);
}
if (!ISEQUALF(ISGetThresholdBinaryResult(&segmentor), 1.0)) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetThresholdBinaryResult failed");
    PBErCatch(PBImgAnalysisErr);
}
if (ISGetSizePool(&segmentor) != GENALG_NBENTITIES) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetSizePool failed");
    PBErCatch(PBImgAnalysisErr);
}
ISSetSizePool(&segmentor, GENALG_NBENTITIES + 100);
if (ISGetSizePool(&segmentor) != GENALG_NBENTITIES + 100) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISSetSizePool failed");
    PBErCatch(PBImgAnalysisErr);
}
if (ISGetNbElite(&segmentor) != GENALG_NBELITES) {
    PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBImgAnalysisErr->_msg, "ISGetNbElite failed");
    PBErCatch(PBImgAnalysisErr);
}

```

```

}
ISSetNbElite(&segmentor, GENALG_NBELITES + 10);
if (ISGetNbElite(&segmentor) != GENALG_NBELITES + 10) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ISSetNbElite failed");
    PBErrCatch(PBIImgAnalysisErr);
}
if (!ISEQUALF(ISGetTargetBestValue(&segmentor), 0.9999)) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ISGetTargetBestValue failed");
    PBErrCatch(PBIImgAnalysisErr);
}
ISSetTargetBestValue(&segmentor, 0.5);
if (!ISEQUALF(ISGetTargetBestValue(&segmentor), 0.5)) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ISSetTargetBestValue failed");
    PBErrCatch(PBIImgAnalysisErr);
}
if (ISGetSizeMaxPool(&segmentor) != segmentor._sizeMaxPool) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "ISGetSizeMaxPool failed");
    PBErrCatch(GenAlgErr);
}
if (ISGetSizeMinPool(&segmentor) != segmentor._sizeMinPool) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "ISGetSizeMinPool failed");
    PBErrCatch(GenAlgErr);
}
ISSetSizeMaxPool(&segmentor, 100);
if (ISGetSizeMaxPool(&segmentor) != 100) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "ISSetSizeMaxPool failed");
    PBErrCatch(GenAlgErr);
}
ISSetSizeMinPool(&segmentor, 100);
if (ISGetSizeMinPool(&segmentor) != 100) {
    GenAlgErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GenAlgErr->_msg, "ISSetSizeMinPool failed");
    PBErrCatch(GenAlgErr);
}
ImgSegmentorFreeStatic(&segmentor);
printf("UnitTestImgSegmentorAddCriterionGetSet OK\n");
}

void UnitTestImgSegmentorSaveLoad() {
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    if (ISAddCriterionRGB(&segmentor, NULL) == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg,
            "UnitTestImgSegmentorSaveLoad failed");
        PBErrCatch(PBIImgAnalysisErr);
    }
    ImgSegmentorCriterionRGB2HSV* criterionHSV =
        ISAddCriterionRGB2HSV(&segmentor, NULL);
    if (criterionHSV == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg,
            "UnitTestImgSegmentorSaveLoad failed");
        PBErrCatch(PBIImgAnalysisErr);
    }
    if (ISAddCriterionRGB(&segmentor, criterionHSV) == NULL) {

```



```

    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg,
        "UnitTestImgSegmentorSaveLoad failed");
    PBErrCatch(PBIImgAnalysisErr);
}
char* fileName = "unitTestImgSegmentorSaveLoad.json";
FILE* stream = fopen(fileName, "w");
if (!ISSave(&segmentor, stream, false)) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorSave failed");
    PBErrCatch(PBIImgAnalysisErr);
}
fclose(stream);
stream = fopen(fileName, "r");
ImgSegmentor load = ImgSegmentorCreateStatic(1);
if (!ISLoad(&load, stream)) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorLoad failed");
    PBErrCatch(PBIImgAnalysisErr);
}
fclose(stream);
if (load._nbClass != segmentor._nbClass ||
    load._flagBinaryResult != segmentor._flagBinaryResult ||
    load._thresholdBinaryResult != segmentor._thresholdBinaryResult ||
    load._nbEpoch != segmentor._nbEpoch ||
    load._sizePool != segmentor._sizePool ||
    load._nbElite != segmentor._nbElite ||
    load._targetBestValue != segmentor._targetBestValue) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorLoad failed");
    PBErrCatch(PBIImgAnalysisErr);
}

if (load._criteria._data != segmentor._criteria._data) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorLoad failed");
    PBErrCatch(PBIImgAnalysisErr);
}
ImgSegmentorCriterion* criteriaA = (ImgSegmentorCriterion*)
    GenTreeData((GenTree*)GSetGet(&(load._criteria._subtrees), 0));
ImgSegmentorCriterion* criteriaB = (ImgSegmentorCriterion*)
    GenTreeData((GenTree*)GSetGet(&(segmentor._criteria._subtrees), 0));
if (criteriaA->_type != criteriaB->_type) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorLoad failed");
    PBErrCatch(PBIImgAnalysisErr);
}
criteriaA = (ImgSegmentorCriterion*)
    GenTreeData((GenTree*)GSetGet(&(load._criteria._subtrees), 1));
criteriaB = (ImgSegmentorCriterion*)
    GenTreeData((GenTree*)GSetGet(&(segmentor._criteria._subtrees), 1));
if (criteriaA->_type != criteriaB->_type) {
    PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
    sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorLoad failed");
    PBErrCatch(PBIImgAnalysisErr);
}
criteriaA = (ImgSegmentorCriterion*)
    GenTreeData((GenTree*)GSetGet(&(((GenTree*)GSetGet(
        &(load._criteria._subtrees), 1))->_subtrees), 0));
criteriaB = (ImgSegmentorCriterion*)
    GenTreeData((GenTree*)GSetGet(&(((GenTree*)GSetGet(
        &(segmentor._criteria._subtrees), 1))->_subtrees), 0));

```

```

    if (criteriaA->_type != criteriaB->_type) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg, "ImgSegmentorLoad failed");
        PBErrCatch(PBIImgAnalysisErr);
    }

    ImgSegmentorFreeStatic(&segmentor);
    ImgSegmentorFreeStatic(&load);
    printf("UnitTestImgSegmentorSaveLoad OK\n");
}

void UnitTestImgSegmentorPredict() {
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    (void)ISAddCriterionRGB(&segmentor, NULL);
    char* fileNameIn = "ISPredict-in.tga";
    char fileNameOut[20];
    GenBrush* img = GBCreateFromFile(fileNameIn);
    GenBrush** res = ISPredict(&segmentor, img);
    for (int iClass = nbClass; iClass--;) {
        sprintf(fileNameOut, "ISPredict-out%02d.tga", iClass);
        GBSetFileName(res[iClass], fileNameOut);
        GBRender(res[iClass]);
    }
    ImgSegmentorFreeStatic(&segmentor);
    for (int iClass = nbClass; iClass--;)
        GBFree(res + iClass);
    free(res);
    GBFree(&img);
    printf("UnitTestImgSegmentorPredict OK\n");
}

void UnitTestImgSegmentorTrain01() {
    srand(2);
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    if (ISAddCriterionRGB(&segmentor, NULL) == NULL) {
        PBIImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBIImgAnalysisErr->_msg, "UnitTestImgSegmentorTrain01 failed");
        PBErrCatch(PBIImgAnalysisErr);
    }
    char* cfgFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", "dataset.json");
    GDataSetGenBrushPair dataSet =
        GDataSetGenBrushPairCreateStatic(cfgFilePath);
    ISSetSizePool(&segmentor, 16);
    ISSetNbElite(&segmentor, 5);
    ISSetSizeMaxPool(&segmentor, 128);
    ISSetSizeMinPool(&segmentor, 16);
    ISSetNbEpoch(&segmentor, 10);
    ISSetTargetBestValue(&segmentor, 0.99);
    ISSetFlagTextOMeter(&segmentor, true);
    ITrain(&segmentor, &dataSet);
    char resFileName[] = "unitTestImgSegmentorTrain01.json";
    FILE* fp = fopen(resFileName, "w");
    if (!ISSave(&segmentor, fp, false)) {
        fprintf(stderr, "Couldn't save %s\n", resFileName);
    }
    fclose(fp);
    fp = fopen(resFileName, "r");
    if (!ISLoad(&segmentor, fp)) {
        fprintf(stderr, "Couldn't load %s\n", resFileName);
    }
}

```

```

}
fclose(fp);
char* imgFilePath = PBFSJoinPath(
    ".", "UnitTestImgSegmentorTrain", "img000.tga");
GenBrush* img = GBCreateFromFile(imgFilePath);
ISSetFlagBinaryResult(&segmentor, true);
GenBrush** pred = ISPredict(&segmentor, img);
for (int iClass = nbClass; iClass--;) {
    char outPath[100];
    sprintf(outPath, "pred000-%03d.tga", iClass);
    char* predFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", outPath);
    GBSetFileName(pred[iClass], predFilePath);
    GBRender(pred[iClass]);
    GBFree(pred + iClass);
    free(predFilePath);
}
free(pred);
GBFree(&img);
free(cfgFilePath);
free(imgFilePath);
GDataSetGenBrushPairFreeStatic(&dataSet);
ImgSegmentorFreeStatic(&segmentor);
printf("UnitTestImgSegmentorTrain01 OK\n");
}

void UnitTestImgSegmentorTrain02() {
    srandom(2);
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    ImgSegmentorCriterionRGB2HSV* criterionHSV =
        ISAddCriterionRGB2HSV(&segmentor, NULL);
    if (criterionHSV == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "UnitTestImgSegmentorTrain02 failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (ISAddCriterionRGB(&segmentor, criterionHSV) == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "UnitTestImgSegmentorTrain02 failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    char* cfgFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", "dataset.json");
    GDataSetGenBrushPair dataSet =
        GDataSetGenBrushPairCreateStatic(cfgFilePath);
    ISSetSizePool(&segmentor, 16);
    ISSetNbElite(&segmentor, 5);
    ISSetSizeMaxPool(&segmentor, 128);
    ISSetSizeMinPool(&segmentor, 16);
    ISSetNbEpoch(&segmentor, 10);
    ISSetTargetBestValue(&segmentor, 0.99);
    ISSetFlagTextOMeter(&segmentor, true);
    ITrain(&segmentor, &dataSet);
    char resFileName[] = "unitTestImgSegmentorTrain02.json";
    FILE* fp = fopen(resFileName, "w");
    if (!ISSave(&segmentor, fp, false)) {
        fprintf(stderr, "Couldn't save %s\n", resFileName);
    }
    fclose(fp);
    fp = fopen(resFileName, "r");
    if (!ISLoad(&segmentor, fp)) {

```

```

    fprintf(stderr, "Couldn't load %s\n", resFileName);
}
fclose(fp);
char* imgFilePath = PBFSJoinPath(
    ".", "UnitTestImgSegmentorTrain", "img001.tga");
GenBrush* img = GBCreateFromFile(imgFilePath);
ISSetFlagBinaryResult(&segmentor, true);
GenBrush** pred = ISPredict(&segmentor, img);
for (int iClass = nbClass; iClass--;) {
    char outPath[100];
    sprintf(outPath, "pred001-%03d.tga", iClass);
    char* predFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", outPath);
    GBSetFileName(pred[iClass], predFilePath);
    GBRender(pred[iClass]);
    GBFree(pred + iClass);
    free(predFilePath);
}
free(pred);
GBFree(&img);
free(cfgFilePath);
free(imgFilePath);
GDataSetGenBrushPairFreeStatic(&dataSet);
ImgSegmentorFreeStatic(&segmentor);
printf("UnitTestImgSegmentorTrain02 OK\n");
}

void UnitTestImgSegmentorTrain03() {
    srand(2);
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    if (ISAddCriterionRGB(&segmentor, NULL) == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "UnitTestImgSegmentorTrain02 failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    ImgSegmentorCriterionRGB2HSV* criterionHSV =
        ISAddCriterionRGB2HSV(&segmentor, NULL);
    if (criterionHSV == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "UnitTestImgSegmentorTrain02 failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    if (ISAddCriterionRGB(&segmentor, criterionHSV) == NULL) {
        PBImpAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImpAnalysisErr->_msg, "UnitTestImgSegmentorTrain02 failed");
        PBErrCatch(PBImpAnalysisErr);
    }
    char* cfgFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", "dataset.json");
    GDataSetGenBrushPair dataSet =
        GDataSetGenBrushPairCreateStatic(cfgFilePath);
    ISSetSizePool(&segmentor, 16);
    ISSetNbElite(&segmentor, 5);
    ISSetSizeMaxPool(&segmentor, 128);
    ISSetSizeMinPool(&segmentor, 16);
    ISSetNbEpoch(&segmentor, 10);
    ISSetTargetBestValue(&segmentor, 0.99);
    ISSetFlagTextOMeter(&segmentor, true);
    ITrain(&segmentor, &dataSet);
    char resFileName[] = "unitTestImgSegmentorTrain03.json";
    FILE* fp = fopen(resFileName, "w");

```

```

if (!ISSave(&segmentor, fp, false)) {
    fprintf(stderr, "Couldn't save %s\n", resFileName);
}
fclose(fp);
fp = fopen(resFileName, "r");
if (!ISLoad(&segmentor, fp)) {
    fprintf(stderr, "Couldn't load %s\n", resFileName);
}
fclose(fp);
char* imgFilePath = PBFSJoinPath(
    ".", "UnitTestImgSegmentorTrain", "img002.tga");
GenBrush* img = GBCreateFromFile(imgFilePath);
ISSetFlagBinaryResult(&segmentor, true);
GenBrush** pred = ISPredict(&segmentor, img);
for (int iClass = nbClass; iClass--;) {
    char outPath[100];
    sprintf(outPath, "pred002-%03d.tga", iClass);
    char* predFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", outPath);
    GBSetFileName(pred[iClass], predFilePath);
    GBRender(pred[iClass]);
    GBFree(pred + iClass);
    free(predFilePath);
}
free(pred);
GBFree(&img);
free(cfgFilePath);
free(imgFilePath);
GDataSetGenBrushPairFreeStatic(&dataSet);
ImgSegmentorFreeStatic(&segmentor);
printf("UnitTestImgSegmentorTrain03 OK\n");
}

void UnitTestImgSegmentorTrain04() {
    srand(2);
    int nbClass = 2;
    ImgSegmentor segmentor = ImgSegmentorCreateStatic(nbClass);
    int rank = 1;
    int size = 2;
    if (ISAddCriterionTex(&segmentor, NULL, rank, size) == NULL) {
        PBImgAnalysisErr->_type = PBErrTypeUnitTestFailed;
        sprintf(PBImgAnalysisErr->_msg, "UnitTestImgSegmentorTrain04 failed");
        PBErrCatch(PBImgAnalysisErr);
    }
    char* cfgFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", "dataset.json");
    GDataSetGenBrushPair dataSet =
        GDataSetGenBrushPairCreateStatic(cfgFilePath);
    ISSetSizePool(&segmentor, 4);
    ISSetNbElite(&segmentor, 2);
    ISSetSizeMaxPool(&segmentor, 4);
    ISSetSizeMinPool(&segmentor, 4);
    ISSetNbEpoch(&segmentor, 2);
    ISSetTargetBestValue(&segmentor, 0.99);
    ISSetFlagTextOMeter(&segmentor, true);
    ITrain(&segmentor, &dataSet);
    char resFileName[] = "unitTestImgSegmentorTrain04.json";
    FILE* fp = fopen(resFileName, "w");
    if (!ISSave(&segmentor, fp, false)) {
        fprintf(stderr, "Couldn't save %s\n", resFileName);
    }
    fclose(fp);
}

```

```

fp = fopen(resFileName, "r");
if (!ISLoad(&segmentor, fp)) {
    fprintf(stderr, "Couldn't load %s\n", resFileName);
}
fclose(fp);
char* imgFilePath = PBFSJoinPath(
    ".", "UnitTestImgSegmentorTrain", "img003.tga");
GenBrush* img = GBCreateFromFile(imgFilePath);
ISSetFlagBinaryResult(&segmentor, true);
GenBrush** pred = ISPredict(&segmentor, img);
for (int iClass = nbClass; iClass--;) {
    char outPath[100];
    sprintf(outPath, "pred003-%03d.tga", iClass);
    char* predFilePath = PBFSJoinPath(
        ".", "UnitTestImgSegmentorTrain", outPath);
    GBSetFileName(pred[iClass], predFilePath);
    GBRender(pred[iClass]);
    GBFree(pred + iClass);
    free(predFilePath);
}
free(pred);
GBFree(&img);
free(cfgFilePath);
free(imgFilePath);
GDataSetGenBrushPairFreeStatic(&dataSet);
ImgSegmentorFreeStatic(&segmentor);
printf("UnitTestImgSegmentorTrain04 OK\n");
}

void UnitTestImgSegmentor() {
    UnitTestImgSegmentorCreateFree();
    UnitTestImgSegmentorAddCriterionGetSet();
    UnitTestImgSegmentorSaveLoad();
    UnitTestImgSegmentorPredict();
    UnitTestImgSegmentorTrain01();
    UnitTestImgSegmentorTrain02();
    UnitTestImgSegmentorTrain03();
    UnitTestImgSegmentorTrain04();
    printf("UnitTestImgSegmentor OK\n");
}

void UnitTestAll() {
    UnitTestImgKMeansClusters();
    UnitTestIntersectionOverUnion();
    UnitTestGBSimilarityCoefficient();
    UnitTestImgSegmentorRGB();
    UnitTestImgSegmentor();
}

int main(void) {
    UnitTestAll();
    return 0;
}

```

5 Unit tests output

```

./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=1:
<190.271,188.622,189.519,255.874>

```

```

<57.922,71.614,92.852,255.544>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=1:
<197.903,195.060,194.940,255.852>
<46.857,55.700,72.989,255.384>
<129.141,141.318,156.154,255.440>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=2 cell=3:
<70.561,84.186,107.671,255.546,66.415,80.028,103.270,255.546,63.722,77.315,100.200,255.546,60.385,74.097,95.695,255.546>
<196.476,194.722,195.635,255.874,194.379,192.612,193.523,255.874,192.848,191.093,192.012,255.874,191.376,189.680,190.680,190.680>
./ImgKMeansClustersTest/imgkmeanscluster.tga size K=3 cell=3:
<203.108,200.359,200.281,255.851,201.244,198.444,198.356,255.851,199.906,197.080,196.987,255.851,198.732,195.894,195.894,195.894>
<58.046,67.067,87.513,255.383,54.053,62.941,83.006,255.383,51.554,60.343,79.943,255.383,48.753,57.583,75.451,255.383>
<142.267,153.344,167.998,255.445,138.511,149.808,164.556,255.445,135.723,147.234,162.003,255.445,132.033,143.971,158.971,158.971>
UnitTestImgKMeansClusters OK
UnitTestIntersectionOverUnion OK
UnitTestIntersectionOverUnion OK
UnitTestImgSegmentorRGB OK
UnitTestImgSegmentorCreateFree OK
UnitTestImgSegmentorAddCriterionGetSet OK
UnitTestImgSegmentorSaveLoad OK
UnitTestImgSegmentorPredict OK
Epoch 00001/00010 TrainAcc[0,1] 0.025461/0.990000 on Sun Apr 28 23:30:17 2019
Epoch 00001/00010 TrainAcc[0,1] 0.354482/0.990000 on Sun Apr 28 23:30:23 2019
Epoch 00002/00010 TrainAcc[0,1] 0.354518/0.990000 on Sun Apr 28 23:31:12 2019
Epoch 00002/00010 TrainAcc[0,1] 0.470892/0.990000 on Sun Apr 28 23:31:19 2019
Epoch 00005/00010 TrainAcc[0,1] 0.511004/0.990000 on Sun Apr 28 23:33:42 2019
Epoch 00006/00010 TrainAcc[0,1] 0.828990/0.990000 on Sun Apr 28 23:37:33 2019
Epoch 00010/00010 TrainAcc[0,1] 0.856932/0.990000 on Sun Apr 28 23:45:28 2019
UnitTestImgSegmentorTrain01 OK
Epoch 00001/00010 TrainAcc[0,1] 0.129935/0.990000 on Sun Apr 28 23:51:50 2019
Epoch 00001/00010 TrainAcc[0,1] 0.131126/0.990000 on Sun Apr 28 23:51:59 2019
Epoch 00001/00010 TrainAcc[0,1] 0.363959/0.990000 on Sun Apr 28 23:52:32 2019
Epoch 00005/00010 TrainAcc[0,1] 0.493795/0.990000 on Sun Apr 28 23:58:43 2019
Epoch 00007/00010 TrainAcc[0,1] 0.500124/0.990000 on Mon Apr 29 00:02:01 2019
Epoch 00009/00010 TrainAcc[0,1] 0.583601/0.990000 on Mon Apr 29 00:11:16 2019
UnitTestImgSegmentorTrain02 OK
Epoch 00001/00010 TrainAcc[0,1] 0.025394/0.990000 on Mon Apr 29 00:13:33 2019
Epoch 00001/00010 TrainAcc[0,1] 0.123240/0.990000 on Mon Apr 29 00:13:41 2019
Epoch 00001/00010 TrainAcc[0,1] 0.214525/0.990000 on Mon Apr 29 00:13:56 2019
Epoch 00001/00010 TrainAcc[0,1] 0.354480/0.990000 on Mon Apr 29 00:14:40 2019
Epoch 00003/00010 TrainAcc[0,1] 0.448067/0.990000 on Mon Apr 29 00:17:49 2019
Epoch 00005/00010 TrainAcc[0,1] 0.720043/0.990000 on Mon Apr 29 00:22:12 2019
Epoch 00008/00010 TrainAcc[0,1] 0.757622/0.990000 on Mon Apr 29 00:27:44 2019
UnitTestImgSegmentorTrain03 OK
Epoch 00001/00002 TrainAcc[0,1] 0.008347/0.990000 on Mon Apr 29 00:47:13 2019
Epoch 00001/00002 TrainAcc[0,1] 0.009338/0.990000 on Mon Apr 29 00:48:42 2019
Epoch 00001/00002 TrainAcc[0,1] 0.023289/0.990000 on Mon Apr 29 00:50:14 2019
UnitTestImgSegmentorTrain04 OK
UnitTestImgSegmentor OK

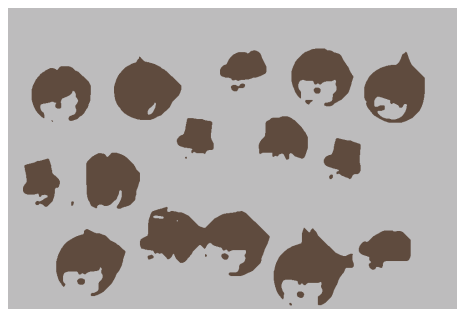
```

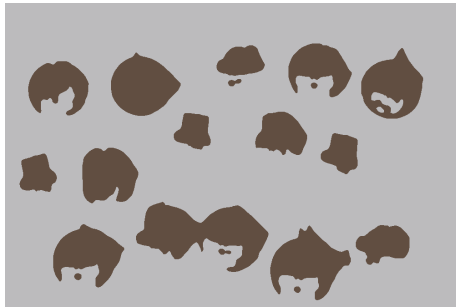
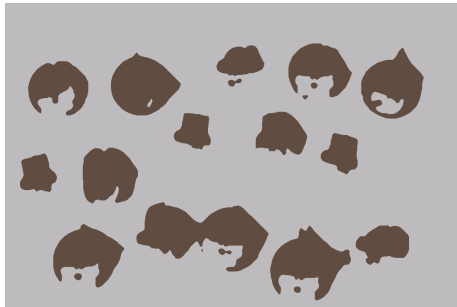
5.1 K-Means clustering on RGBA space

imgkmeanscluster.tga:

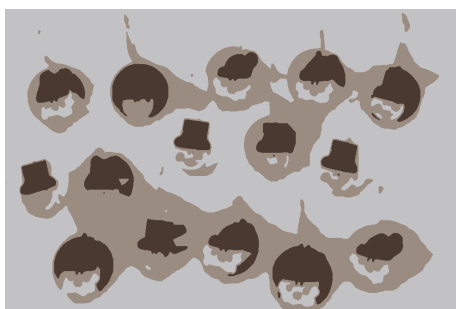
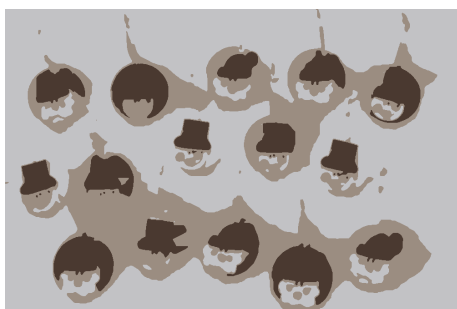
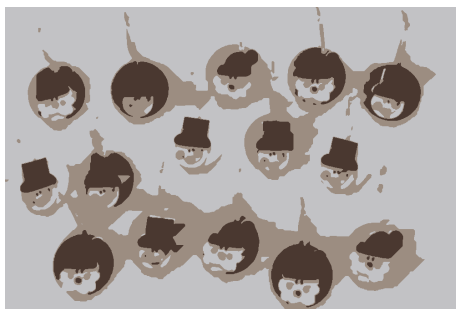


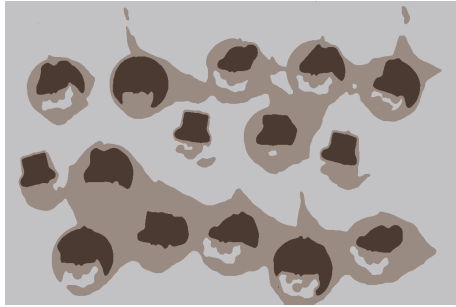
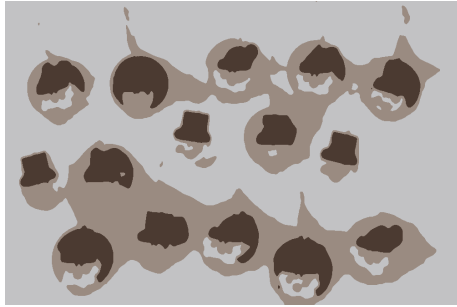
clustering for K equals 2 to 6 and radius equals 0 to 5:
K=2:





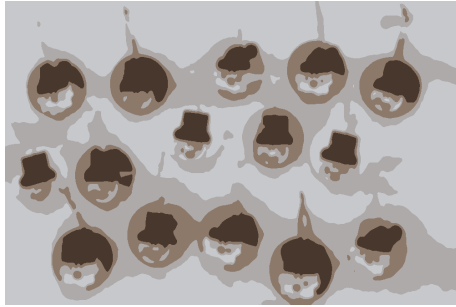
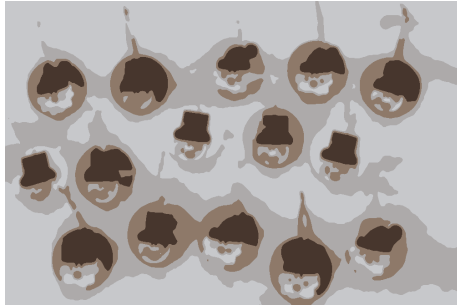
K=3:





K=4:





K=5:





K=6:





imgkmeanscluster.txt:

```
{
  "_size": "1",
  "_clusters": {
    "_seed": "1",
    "_centers": [
      {
        "_dim": "36",
        "_val": ["203.108261", "200.358521", "200.281174", "255.851440", "201.244080", "198.444351", "198.355515", "255.851440"],
      },
      {
        "_dim": "36",
        "_val": ["58.045620", "67.067284", "87.513313", "255.382996", "54.052647", "62.940845", "83.005745", "255.382996"],
      },
      {
        "_dim": "36",
        "_val": ["142.266953", "153.344070", "167.997711", "255.444534", "138.511063", "149.808060", "164.556274", "255.444534"],
      }
    ]
  }
}
```

5.2 ImgSegmentor

5.2.1 Test 01

unitTestImgSegmentorTrain01.json:

```
{
  "_nbClass": "2",
  "_flagBinaryResult": "0",
  "_thresholdBinaryResult": "0.500000",
  "_nbEpoch": "10",
  "_sizePool": "16",
  "_nbElite": "5",
  "_targetBestValue": "0.990000",
  "_criteria": {
    "_nbSubtree": "1",
    "_subtree_0": {
      "_criterion": {
        "_type": "0",

```

```

    "_nbClass": "2",
    "_neuranet": {
      "_nbInputVal": "3",
      "_nbOutputVal": "2",
      "_nbMaxHidVal": "18",
      "_nbMaxBases": "90",
      "_nbMaxLinks": "90",
      "_bases": {
        "_dim": "270",
        "_val": ["-0.627757", "-0.528276", "0.431686", "0.173828", "0.936959", "0.139028", "-0.680593", "0.333254", "0.98",
      ],
      "_links": {
        "_dim": "270",
        "_val": ["0", "0", "3", "1", "0", "4", "2", "0", "5", "3", "0", "6", "4", "0", "7", "5", "0", "8", "6", "0", "9", "7", "0", "10",
      ]
    }
  },
  "_nbSubtree": "0"
}
}
}

```



5.2.2 Test 02

unitTestImgSegmentorTrain02.json:

```

{
  "_nbClass": "2",
  "_flagBinaryResult": "0",
  "_thresholdBinaryResult": "0.500000",
  "_nbEpoch": "10",
  "_sizePool": "16",
  "_nbElite": "5",
  "_targetBestValue": "0.990000",
  "_criteria": {
    "_nbSubtree": "1",
    "_subtree_0": {
      "_criterion": {
        "_type": "1",
        "_nbClass": "2"
      }
    },
    "_nbSubtree": "1",
    "_subtree_0": {
      "_criterion": {
        "_type": "0",
        "_nbClass": "2",
        "_neuranet": {
          "_nbInputVal": "3",

```