GDataSet

P. Baillehache

February 10, 2019

Contents

Introduction

GDataSet is a C library to manipulate generic data sets.

It offers the following functionalities:

- loading a data set from its description file

- splitting the data set into user defined categories (e.g. training, validation, test)

- shuffling the data set

- looping through the samples of the data set.

It provides an unique interface to several implementation supporting various types of dataset. Supported types are: VecFloat and pair of GenBrush (img/mask).

The GDataSet library uses the `PBErr`, `GSet`, `PBJson`, `PBMath` and `PBFileSys` libraries.

1   Interface

```
// ============= GDATASET_H ================

#ifndef GDATASET_H
#define GDATASET_H

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

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <execinfo.h>
#include <errno.h>
#include <string.h>
#include "pberr.h"
#include "gset.h"
#include "pbmath.h"
#include "pbjson.h"
#include "pbfilesys.h"

// Define locally the needed types and functions for libraries that were
// not included to allow the user to inlcude only what's needed for her
// application
#ifndef GENBRUSH_H
typedef struct GenBrush GenBrush;
typedef enum GBScaleMethod {GBScaleMethod_Default} GBScaleMethod;
GenBrush* GBCreateFromFile(const char* const fileName);
GenBrush* GBScale(const GenBrush* const that,
  const VecShort2D* const dim, const GBScaleMethod scaleMethod);
void GBFree(GenBrush** that);
VecShort2D* GBDim(const GenBrush* const that);
#endif

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

typedef enum GDataSetType {
  GDataSetType_VecFloat, GDataSetType_GenBrushPair
} GDataSetType;

// ================= Data structures ===================

typedef struct GDataSet {
  // Path to the config file of the data set
  char* _cfgFilePath;
  // Data in the config file encoded as JSON object
```

```
  JSONNode* _json;
  // Name of the data set
  char* _name;
  // Description of the data set
  char* _desc;
  // Type of set
  GDataSetType _type;
  // Nb of samples
  int _nbSample;
  // Set of samples
  GSet _samples;
  // Dimensions of each sample, they must have all the same dimension
  // e.g.:
  // if samples are VecFloat<3> then _dim = VecShort<1>[3]
  // if samples are GenBrush then _dim = VecShort<2>[width, height]
  VecShort* _sampleDim;
  // Splitting of samples
  VecShort* _split;
  // Sets of splitted samples
  GSet* _categories;
  // Iterators on the sets of splitted samples
  GSetIterForward* _iterators;
} GDataSet;

typedef struct GDataSetVecFloat {
  // Generic GDataSet
  GDataSet _dataSet;
} GDataSetVecFloat;

typedef struct GDataSetGenBrushPair {
  // Generic GDataSet
  GDataSet _dataSet;
  // Format of images
  char* _format;
  // Dimensions of images
  VecShort2D _dim;
} GDataSetGenBrushPair;

typedef struct GDSFilePathPair {
  char* _path[2];
} GDSFilePathPair;

typedef struct GDSGenBrushPair {
  GenBrush* _img;
  GenBrush* _mask;
} GDSGenBrushPair;

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

// Create a new GDataSet defined by the file at 'cfgFilePath'
// The random generator must have been initialized before calling
// this function
GDataSet GDataSetCreateStatic(const char* const cfgFilePath);

// Free the memory used by a GDataSet
void GDataSetFreeStatic(GDataSet* const that);

// Create a new GDataSetVecFloat defined by the file at 'cfgFilePath'
// The random generator must have been initialized before calling
// this function
GDataSetVecFloat GDataSetVecFloatCreateStatic(
  const char* const cfgFilePath);
```

```
// Free the memory used by a GDataSetVecFloat
void GDataSetVecFloatFreeStatic(GDataSetVecFloat* const that);

// Create a new GDataSetGenBrushPair defined by the file at 'cfgFilePath'
// The random generator must have been initialized before calling
// this function
GDataSetGenBrushPair GDataSetGenBrushPairCreateStatic(
  const char* const cfgFilePath);

// Free the memory used by a GDataSetGenBrushPair
void GDataSetGenBrushPairFreeStatic(GDataSetGenBrushPair* const that);

// Get the total number of samples in the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
long _GDSGetSize(const GDataSet* const that);

// Get the total number of samples in the GDataSet 'that' for the
// category 'iCat'. Return 0 if the category doesn't exists
#if BUILDMODE != 0
inline
#endif
long _GDSGetSizeCat(const GDataSet* const that, const long iCat);

// Split the samples of the GDataSet 'that' into several categories
// defined by 'cat'. The dimension of 'cat' gives the number of
// categories and the value for each dimension of 'cat' gives the
// number of samples in the corresponding category. For example <3,4>
// would mean 2 categories with 3 samples in the first one and 4
// samples in the second one. There must me at least as many samples
// in the data set as the sum of samples in 'cat'.
// Each category must have at least one sample. Samples are allocated // randomly to the categories.
// If 'that' was already splitted the previous splitting is discarded.
void _GDSSplit(GDataSet* const that, const VecShort* const cat);

// Unsplit the GDataSet 'that', i.e. after calling GDataSetUnsplit 'that'
// has only one category containing all the samples
#if BUILDMODE != 0
inline
#endif
void _GDSUnsplit(GDataSet* const that);

// Shuffle the samples of the category 'iCat' of the GDataSet 'that'.
// Reset the iterator of the category
#if BUILDMODE != 0
inline
#endif
void _GDSShuffle(GDataSet* const that, const long iCat);

// Shuffle the samples of all the categories of the GDataSet 'that'.
// Reset the iterator of the categories
#if BUILDMODE != 0
inline
#endif
void _GDSShuffleAll(GDataSet* const that);

// Get the name of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
```

```
const char* _GDSName(const GDataSet* const that);

// Get the description of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const char* _GDSDesc(const GDataSet* const that);

// Get the path of the config file of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const char* _GDSCfgFilePath(const GDataSet* const that);

// Get a copy of the path of the config file of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
char* _GDSGetCfgFilePath(const GDataSet* const that);

// Get the path of the folder of the config file of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
char* _GDSGetCfgFolderPath(const GDataSet* const that);

// Get the type of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
GDataSetType _GDSGetType(const GDataSet* const that);

// Get the number of categories of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
long _GDSGetNbCat(const GDataSet* const that);

// If there is a next sample move to the next sample of the category
// 'iCat' and return true, else return false
#if BUILDMODE != 0
inline
#endif
bool _GDSStepSample(const GDataSet* const that, const long iCat);

// Reset the iterator on category 'iCat' of the GDataSet 'that', i.e.
// the next call to GDataSetGetNextSample will give the first sample of
// the category 'iCat'
#if BUILDMODE != 0
inline
#endif
void _GDSReset(GDataSet* const that, const long iCat);

// Reset the iterator on all categories of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
void _GDSResetAll(GDataSet* const that);

// Get the current sample in the category 'iCat' of the GDataSet 'that'
void* _GDSGetSample(
  const GDataSet* const that, const int iCat);
```

```c
VecFloat* GDSGetSampleVecFloat(
  const GDataSetVecFloat* const that, const int iCat);
GDSGenBrushPair* GDSGetSampleGenBrushPair(
  const GDataSetGenBrushPair* const that, const int iCat);

// Release the memory used by the FilePathPair 'that'
void GDSFilePathPairFree(GDSFilePathPair** const that);

#ifdef GENBRUSH_H
// Release the memory used by the GenBrushPair 'that'
void GDSGenBrushPairFree(GDSGenBrushPair** const that);
#endif
// Get the dimensions of the samples of GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const VecShort* _GDSSampleDim(const GDataSet* const that);

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

#define GDSCfgFilePath(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSCfgFilePath, \
  const GDataSet*: _GDSCfgFilePath, \
  GDataSetVecFloat*: _GDSCfgFilePath, \
  const GDataSetVecFloat*: _GDSCfgFilePath, \
  GDataSetGenBrushPair*: _GDSCfgFilePath, \
  const GDataSetGenBrushPair*: _GDSCfgFilePath, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSGetCfgFilePath(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSGetCfgFilePath, \
  const GDataSet*: _GDSGetCfgFilePath, \
  GDataSetVecFloat*: _GDSGetCfgFilePath, \
  const GDataSetVecFloat*: _GDSGetCfgFilePath, \
  GDataSetGenBrushPair*: _GDSGetCfgFilePath, \
  const GDataSetGenBrushPair*: _GDSGetCfgFilePath, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSGetCfgFolderPath(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSGetCfgFolderPath, \
  const GDataSet*: _GDSGetCfgFolderPath, \
  GDataSetVecFloat*: _GDSGetCfgFolderPath, \
  const GDataSetVecFloat*: _GDSGetCfgFolderPath, \
  GDataSetGenBrushPair*: _GDSGetCfgFolderPath, \
  const GDataSetGenBrushPair*: _GDSGetCfgFolderPath, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSDesc(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSDesc, \
  const GDataSet*: _GDSDesc, \
  GDataSetVecFloat*: _GDSDesc, \
  const GDataSetVecFloat*: _GDSDesc, \
  GDataSetGenBrushPair*: _GDSDesc, \
  const GDataSetGenBrushPair*: _GDSDesc, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSGetNbCat(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSGetNbCat, \
  const GDataSet*: _GDSGetNbCat, \
  GDataSetVecFloat*: _GDSGetNbCat, \
  const GDataSetVecFloat*: _GDSGetNbCat, \
  GDataSetGenBrushPair*: _GDSGetNbCat, \
```

```
  const GDataSetGenBrushPair*: _GDSGetNbCat, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSGetSample(DataSet, ICat) _Generic(DataSet, \
  GDataSet*: _GDSGetSample, \
  const GDataSet*: _GDSGetSample, \
  GDataSetVecFloat*: GDSGetSampleVecFloat, \
  const GDataSetVecFloat*: GDSGetSampleVecFloat, \
  GDataSetGenBrushPair*: GDSGetSampleGenBrushPair, \
  const GDataSetGenBrushPair*: GDSGetSampleGenBrushPair, \
  default: PBErrInvalidPolymorphism)(DataSet, ICat)

#define GDSGetSize(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSGetSize, \
  const GDataSet*: _GDSGetSize, \
  GDataSetVecFloat*: _GDSGetSize, \
  const GDataSetVecFloat*: _GDSGetSize, \
  GDataSetGenBrushPair*: _GDSGetSize, \
  const GDataSetGenBrushPair*: _GDSGetSize, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSGetSizeCat(DataSet, ICat) _Generic(DataSet, \
  GDataSet*: _GDSGetSizeCat, \
  const GDataSet*: _GDSGetSizeCat, \
  GDataSetVecFloat*: _GDSGetSizeCat, \
  const GDataSetVecFloat*: _GDSGetSizeCat, \
  GDataSetGenBrushPair*: _GDSGetSizeCat, \
  const GDataSetGenBrushPair*: _GDSGetSizeCat, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet, ICat)

#define GDSGetType(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSGetType, \
  const GDataSet*: _GDSGetType, \
  GDataSetVecFloat*: _GDSGetType, \
  const GDataSetVecFloat*: _GDSGetType, \
  GDataSetGenBrushPair*: _GDSGetType, \
  const GDataSetGenBrushPair*: _GDSGetType, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSName(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSName, \
  const GDataSet*: _GDSName, \
  GDataSetVecFloat*: _GDSName, \
  const GDataSetVecFloat*: _GDSName, \
  GDataSetGenBrushPair*: _GDSName, \
  const GDataSetGenBrushPair*: _GDSName, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSReset(DataSet, ICat) _Generic(DataSet, \
  GDataSet*: _GDSReset, \
  const GDataSet*: _GDSReset, \
  GDataSetVecFloat*: _GDSReset, \
  const GDataSetVecFloat*: _GDSReset, \
  GDataSetGenBrushPair*: _GDSReset, \
  const GDataSetGenBrushPair*: _GDSReset, \
  default: PBErrInvalidPolymorphism)((GDataSet*)DataSet, ICat)

#define GDSResetAll(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSResetAll, \
  const GDataSet*: _GDSResetAll, \
  GDataSetVecFloat*: _GDSResetAll, \
  const GDataSetVecFloat*: _GDSResetAll, \
```

```
  GDataSetGenBrushPair*: _GDSResetAll, \
  const GDataSetGenBrushPair*: _GDSResetAll, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSSampleDim(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSSampleDim, \
  const GDataSet*: _GDSSampleDim, \
  GDataSetVecFloat*: _GDSSampleDim, \
  const GDataSetVecFloat*: _GDSSampleDim, \
  GDataSetGenBrushPair*: _GDSSampleDim, \
  const GDataSetGenBrushPair*: _GDSSampleDim, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet)

#define GDSShuffle(DataSet, ICat) _Generic(DataSet, \
  GDataSet*: _GDSShuffle, \
  GDataSetVecFloat*: _GDSShuffle, \
  GDataSetGenBrushPair*: _GDSShuffle, \
  default: PBErrInvalidPolymorphism)((GDataSet*)DataSet, ICat)

#define GDSShuffleAll(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSShuffleAll, \
  GDataSetVecFloat*: _GDSShuffleAll, \
  GDataSetGenBrushPair*: _GDSShuffleAll, \
  default: PBErrInvalidPolymorphism)((GDataSet*)DataSet)

#define GDSSplit(DataSet, Cat) _Generic(DataSet, \
  GDataSet*: _GDSSplit, \
  GDataSetVecFloat*: _GDSSplit, \
  GDataSetGenBrushPair*: _GDSSplit, \
  default: PBErrInvalidPolymorphism)((GDataSet*)DataSet, Cat)

#define GDSStepSample(DataSet, ICat) _Generic(DataSet, \
  GDataSet*: _GDSStepSample, \
  const GDataSet*: _GDSStepSample, \
  GDataSetVecFloat*: _GDSStepSample, \
  const GDataSetVecFloat*: _GDSStepSample, \
  GDataSetGenBrushPair*: _GDSStepSample, \
  const GDataSetGenBrushPair*: _GDSStepSample, \
  default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet, ICat)

#define GDSUnsplit(DataSet) _Generic(DataSet, \
  GDataSet*: _GDSUnsplit, \
  GDataSetVecFloat*: _GDSUnsplit, \
  GDataSetGenBrushPair*: _GDSUnsplit, \
  default: PBErrInvalidPolymorphism)((GDataSet*)DataSet)


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

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

#endif
```

2 Code

2.1 gdataset.c

```
// =========== GDATASET_C ===============

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

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

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

// Create a new GDataSet defined by the file at 'cfgFilePath'
// The random generator must have been initialized before calling
// this function
GDataSet GDataSetCreateStatic(const char* const cfgFilePath) {
#if BUILDMODE == 0
  if (cfgFilePath == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'cfgFilePath' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare the new GDataSet
  GDataSet that;
  // Copy the file path
  that._cfgFilePath = PBErrMalloc(GDataSetErr, strlen(cfgFilePath) + 1);
  strcpy(that._cfgFilePath, cfgFilePath);
  // Open the description file
  FILE* cfgFile = fopen(cfgFilePath, "r");
  // If the description file doesn't exist
  if (cfgFile == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(GDataSetErr->_msg, "Can't open the configuration file %s",
      cfgFilePath);
    PBErrCatch(GDataSetErr);
  }
  // Load the encoded data
  that._json = JSONCreate();
  // Load the whole encoded data
  if (JSONLoad(that._json, cfgFile) == false) {
    printf("%s\n", GDataSetErr->_msg);
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg, "Can't load the configuration file");
    PBErrCatch(GDataSetErr);
  }
  // Decode dataSet
  JSONNode* prop = JSONProperty(that._json, "dataSet");
  if (prop == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (dataSet missing)");
    PBErrCatch(GDataSetErr);
  }
  JSONNode* val = JSONValue(prop, 0);
  that._name = PBErrMalloc(GDataSetErr,
    sizeof(char) * (strlen(JSONLabel(val)) + 1));
  strcpy(that._name, JSONLabel(val));
```

```c
  // Decode desc
  prop = JSONProperty(that._json, "desc");
  if (prop == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (desc missing)");
    PBErrCatch(GDataSetErr);
  }
  val = JSONValue(prop, 0);
  that._desc = PBErrMalloc(GDataSetErr,
    sizeof(char) * (strlen(JSONLabel(val)) + 1));
  strcpy(that._desc, JSONLabel(val));
  // Decode dataSetType
  prop = JSONProperty(that._json, "dataSetType");
  if (prop == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (dataSetType missing)");
    PBErrCatch(GDataSetErr);
  }
  val = JSONValue(prop, 0);
  that._type = atoi(JSONLabel(val));
  // Decode dim
  prop = JSONProperty(that._json, "dim");
  if (prop == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (dim missing)");
    PBErrCatch(GDataSetErr);
  }
  that._sampleDim = NULL;
  VecDecodeAsJSON(&(that._sampleDim), prop);
  // Decode nbSample
  prop = JSONProperty(that._json, "nbSample");
  if (prop == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (nbSample missing)");
    PBErrCatch(GDataSetErr);
  }
  val = JSONValue(prop, 0);
  that._nbSample = atoi(JSONLabel(val));
  // Init the splits and iterators
  that._split = NULL;
  that._categories = NULL;
  that._iterators = NULL;

  // Close the description file
  fclose(cfgFile);

  // Return the new GDataSet
  return that;
}

// Free the memory used by a GDataSet
void GDataSetFreeStatic(GDataSet* const that) {
  if (that == NULL)
    return;
  // Free memory
  JSONFree(&(that->_json));
  free(that->_name);
  free(that->_desc);
```

```c
    free(that->_cfgFilePath);
    for (int iCat = GDSGetNbCat(that); iCat--;) {
      GSetFlush(that->_categories + iCat);
    }
    free(that->_categories);
    free(that->_iterators);
    VecFree(&(that->_split));
    VecFree(&(that->_sampleDim));
}

// Create a new GDataSetVecFloat defined by the file at 'cfgFilePath'
// The random generator must have been initialized before calling
// this function
GDataSetVecFloat GDataSetVecFloatCreateStatic(
  const char* const cfgFilePath) {
#if BUILDMODE == 0
  if (cfgFilePath == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'cfgFilePath' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Declare the new GDataSetVecFloat
  GDataSetVecFloat that;
  // Create the generic GDataSet
  that._dataSet = GDataSetCreateStatic(cfgFilePath);
  // Check the type
  if (GDSGetType(&that) != GDataSetType_VecFloat) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg, "Invalid type");
    PBErrCatch(GDataSetErr);
  }
  // Check the samples' dimension
  if (VecGetDim(GDSSampleDim(&that)) != 1) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg, "Invalid sample dimension");
    PBErrCatch(GDataSetErr);
  }
  // Load the samples
  JSONNode* prop = JSONProperty(that._dataSet._json, "samples");
  if (prop == NULL) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (samples missing)");
    PBErrCatch(GDataSetErr);
  }
  if (JSONGetNbValue(prop) != that._dataSet._nbSample) {
    GDataSetErr->_type = PBErrTypeInvalidData;
    sprintf(GDataSetErr->_msg,
      "Invalid description file (samples's number != nbSample)");
    PBErrCatch(GDataSetErr);
  }
  that._dataSet._samples = GSetCreateStatic();
  for (int iSample = 0; iSample < that._dataSet._nbSample; ++iSample) {
    JSONNode* val = JSONValue(prop, iSample);
    VecFloat* v = NULL;
    VecDecodeAsJSON(&v, val);
    GSetAppend(&(that._dataSet._samples), v);
  }
  // Create the initial category
  that._dataSet._split = VecShortCreate(1);
  VecSet(that._dataSet._split, 0, that._dataSet._nbSample);
```

```
    that._dataSet._categories = PBErrMalloc(GDataSetErr, sizeof(GSet));
    that._dataSet._categories[0] = GSetCreateStatic();
    GSetIterForward iter =
      GSetIterForwardCreateStatic(&(that._dataSet._samples));
    do {
      void* sample = GSetIterGet(&iter);
      GSetAppend(that._dataSet._categories, sample);
    } while (GSetIterStep(&iter));
    that._dataSet._iterators =
      PBErrMalloc(GDataSetErr, sizeof(GSetIterForward));
    that._dataSet._iterators[0] =
      GSetIterForwardCreateStatic(that._dataSet._categories);
    // Return the new GDataSetVecFloat
    return that;
}


// Free the memory used by a GDataSetVecFloat
void GDataSetVecFloatFreeStatic(GDataSetVecFloat* const that) {
    if (that == NULL)
      return;
    // Free memory
    GDataSetFreeStatic((GDataSet*)that);
    while (GSetNbElem(&(((GDataSet*)that)->_samples)) > 0) {
      VecFloat* sample = GSetPop(&(((GDataSet*)that)->_samples));
      VecFree(&sample);
    }
}


// Create a new GDataSetGenBrushPair defined by the file at 'cfgFilePath'
// The random generator must have been initialized before calling
// this function
GDataSetGenBrushPair GDataSetGenBrushPairCreateStatic(
  const char* const cfgFilePath) {
#if BUILDMODE == 0
    if (cfgFilePath == NULL) {
      GDataSetErr->_type = PBErrTypeNullPointer;
      sprintf(PBImgAnalysisErr->_msg, "'cfgFilePath' is null");
      PBErrCatch(PBImgAnalysisErr);
    }
#endif
    // Declare the new GDataSetVecFloat
    GDataSetGenBrushPair that;
    // Create the generic GDataSet
    that._dataSet = GDataSetCreateStatic(cfgFilePath);
    // Check the type
    if (GDSGetType(&that) != GDataSetType_GenBrushPair) {
      GDataSetErr->_type = PBErrTypeInvalidData;
      sprintf(GDataSetErr->_msg, "Invalid type");
      PBErrCatch(GDataSetErr);
    }
    // Check the samples' dimension
    if (VecGetDim(GDSSampleDim(&that)) != 2) {
      GDataSetErr->_type = PBErrTypeInvalidData;
      sprintf(GDataSetErr->_msg, "Invalid sample dimension (%ld=2)",
        VecGetDim(GDSSampleDim(&that)));
      PBErrCatch(GDataSetErr);
    }
    // Load the samples
    JSONNode* prop = JSONProperty(that._dataSet._json, "samples");
    if (prop == NULL) {
      GDataSetErr->_type = PBErrTypeInvalidData;
      sprintf(GDataSetErr->_msg,
```

```
        "Invalid description file (samples missing)");
      PBErrCatch(GDataSetErr);
    }
    if (JSONGetNbValue(prop) != that._dataSet._nbSample) {
      GDataSetErr->_type = PBErrTypeInvalidData;
      sprintf(GDataSetErr->_msg,
        "Invalid description file (samples's number != nbSample)");
      PBErrCatch(GDataSetErr);
    }
    that._dataSet._samples = GSetCreateStatic();
    for (int iSample = 0; iSample < that._dataSet._nbSample; ++iSample) {
      JSONNode* val = JSONValue(prop, iSample);
      // Allocate memory for the pair image/mask
      GDSFilePathPair* pair = PBErrMalloc(GDataSetErr,
        sizeof(GDSFilePathPair));
      // Decode img
      JSONNode* subProp = JSONProperty(val, "img");
      if (subProp == NULL) {
        GDataSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg,
          "Invalid description file (samples.img missing)");
        PBErrCatch(GDataSetErr);
      }
      JSONNode* subVal = JSONValue(subProp, 0);
      pair->_path[0] = PBErrMalloc(GDataSetErr,
        sizeof(char) * (strlen(JSONLabel(subVal)) + 1));
      strcpy(pair->_path[0], JSONLabel(subVal));
      // Decode mask
      subProp = JSONProperty(val, "mask");
      if (subProp == NULL) {
        GDataSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg,
          "Invalid description file (samples.mask missing)");
        PBErrCatch(GDataSetErr);
      }
      subVal = JSONValue(subProp, 0);
      pair->_path[1] = PBErrMalloc(GDataSetErr,
        sizeof(char) * (strlen(JSONLabel(subVal)) + 1));
      strcpy(pair->_path[1], JSONLabel(subVal));
      // Add the pair to the samples
      GSetAppend(&(that._dataSet._samples), pair);
    }
    // Create the initial category
    that._dataSet._split = VecShortCreate(1);
    VecSet(that._dataSet._split, 0, that._dataSet._nbSample);
    that._dataSet._categories = PBErrMalloc(GDataSetErr, sizeof(GSet));
    that._dataSet._categories[0] = GSetCreateStatic();
    GSetIterForward iter =
      GSetIterForwardCreateStatic(&(that._dataSet._samples));
    do {
      void* sample = GSetIterGet(&iter);
      GSetAppend(that._dataSet._categories, sample);
    } while (GSetIterStep(&iter));
    that._dataSet._iterators =
      PBErrMalloc(GDataSetErr, sizeof(GSetIterForward));
    that._dataSet._iterators[0] =
      GSetIterForwardCreateStatic(that._dataSet._categories);
    // Return the new GDataSetVecFloat
    return that;
}


// Free the memory used by a GDataSetGenBrushPair
```

```
void GDataSetGenBrushPairFreeStatic(GDataSetGenBrushPair* const that) {
  if (that == NULL)
    return;
  // Free memory
  GDataSetFreeStatic((GDataSet*)that);
  while (GSetNbElem(&(((GDataSet*)that)->_samples)) > 0) {
    GDSFilePathPair* sample = GSetPop(&(((GDataSet*)that)->_samples));
    GDSFilePathPairFree(&sample);
  }
}

// Split the samples of the GDataSet 'that' into several categories
// defined by 'cat'. The dimension of 'cat' gives the number of
// categories and the value for each dimension of 'cat' gives the
// number of samples in the corresponding category. For example <3,4>
// would mean 2 categories with 3 samples in the first one and 4
// samples in the second one. There must me at least as many samples
// in the data set as the sum of samples in 'cat'.
// Each category must have at least one sample. Samples are allocated // randomly to the categories.
// If 'that' was already splitted the previous splitting is discarded.
void _GDSSplit(GDataSet* const that, const VecShort* const cat) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  long nb = 0;
  for (long iCat = VecGetDim(cat); iCat--;)
    nb += VecGet(cat, iCat);
  if (nb > GDSGetSize(that)) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg,
      "Not enough samples for the requested splitting (%ld<%ld)",
        nb, GDSGetSize(that));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Free the current splitting if necessary
  if (that->_categories != NULL) {
    if (that->_split != NULL) {
      for (int iCat = GDSGetNbCat(that); iCat--;) {
        GSetFlush(that->_categories + iCat);
      }
    }
    free(that->_categories);
  }
  if (that->_iterators)
    free(that->_iterators);
  VecFree(&(that->_split));
  // Get the number of categories
  long nbCat = VecGetDim(cat);
  // Allocate memory for the categories
  that->_categories = PBErrMalloc(GDataSetErr, sizeof(GSet) * nbCat);
  for (long iCat = nbCat; iCat--;) {
    that->_categories[iCat] = GSetCreateStatic();
  }
  // Copy the splitting
  that->_split = VecClone(cat);
  // Shuffle the samples
  GSetShuffle(&(that->_samples));
  // Declare an iterator on the samples
```

```c
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_samples));
    // Loop on categories
    for (long iCat = nbCat; iCat--;) {
      // Get the nb of samples for this category
      long nbSample = VecGet(cat, iCat);
      // Loop on the sample
      for (long iSample = nbSample; iSample--; GSetIterStep(&iter)) {
        // Get the next sample
        void* sample = GSetIterGet(&iter);
        // Add the sample to the category
        GSetAppend(that->_categories + iCat, sample);
      }
    }
    // Allocate memory for the iterators
    that->_iterators = PBErrMalloc(GDataSetErr,
      sizeof(GSetIterForward) * nbCat);
    for (long iCat = nbCat; iCat--;) {
      that->_iterators[iCat] =
        GSetIterForwardCreateStatic(that->_categories + iCat);
    }
}

// Get the current sample in the category 'iCat' of the GDataSet 'that'
void* _GDSGetSample(
  const GDataSet* const that, const int iCat) {
  // Call the appropriate function according to the type
  switch (GDSGetType(that)) {
    case GDataSetType_VecFloat:
      return GDSGetSampleVecFloat((GDataSetVecFloat*)that, iCat);
      break;
    case GDataSetType_GenBrushPair:
      return GDSGetSampleGenBrushPair((GDataSetGenBrushPair*)that, iCat);
      break;
    default:
      return NULL;
      break;
  }
}

VecFloat* GDSGetSampleVecFloat(
  const GDataSetVecFloat* const that, const int iCat) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (iCat < 0 || iCat >= GDSGetNbCat(that)) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'iCat' is invalid (0<=%d<%ld)",
      iCat, GDSGetNbCat(that));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  VecFloat* sample = GSetIterGet(((GDataSet*)that)->_iterators + iCat);
  return VecClone(sample);
}

GDSGenBrushPair* GDSGetSampleGenBrushPair(
  const GDataSetGenBrushPair* const that, const int iCat) {
#if BUILDMODE == 0
  if (that == NULL) {
```

```c
      GDataSetErr->_type = PBErrTypeNullPointer;
      sprintf(PBImgAnalysisErr->_msg, "'that' is null");
      PBErrCatch(PBImgAnalysisErr);
    }
    if (iCat < 0 || iCat >= GDSGetNbCat(that)) {
      GDataSetErr->_type = PBErrTypeInvalidArg;
      sprintf(PBImgAnalysisErr->_msg, "'iCat' is invalid (0<=%d<%ld)",
        iCat, GDSGetNbCat(that));
      PBErrCatch(PBImgAnalysisErr);
    }
#endif
  GDSFilePathPair* pairFile =
    GSetIterGet(((GDataSet*)that)->_iterators + iCat);
  GDSGenBrushPair* pairSample = PBErrMalloc(GDataSetErr,
    sizeof(GDSGenBrushPair));
  char* root = GDSGetCfgFolderPath(that);
  char* path = PBFSJoinPath(root, pairFile->_path[0]);
  GenBrush* gb = GBCreateFromFile(path);
  // Rescale the sample if needed to always provide to the user
  // the dimensions defined in the configuration file of the data set
  if (!VecIsEqual(GBDim(gb), GDSSampleDim(that))) {
    pairSample->_img = GBScale(gb,
      (const VecShort2D*)GDSSampleDim(that), GBScaleMethod_Default);
    GBFree(&gb);
  } else {
    pairSample->_img = gb;
  }
  free(path);
  path = PBFSJoinPath(root, pairFile->_path[1]);
  gb = GBCreateFromFile(path);
  if (!VecIsEqual(GBDim(gb), GDSSampleDim(that))) {
    pairSample->_mask = GBScale(gb,
      (const VecShort2D*)GDSSampleDim(that), GBScaleMethod_Default);
    GBFree(&gb);
  } else {
    pairSample->_mask = gb;
  }
  free(path);
  free(root);
  return pairSample;
}

// Release the memory used by the FilePathPair 'that'
void GDSFilePathPairFree(GDSFilePathPair** const that) {
  if (that == NULL || *that == NULL)
    return;
  free((*that)->_path[0]);
  free((*that)->_path[1]);
  free(*that);
  *that = NULL;
}

// Release the memory used by the GenBrushPair 'that'
void GDSGenBrushPairFree(GDSGenBrushPair** const that) {
  if (that == NULL || *that == NULL)
    return;
  GBFree(&((*that)->_img));
  GBFree(&((*that)->_mask));
  free(*that);
  *that = NULL;
}
```

2.2 gdataset-inline.c

```
// ============= GDATASET_INLINE.C ================

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

// Get the total number of samples in the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
long _GDSGetSize(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return that->_nbSample;
}

// Get the total number of samples in the GDataSet 'that' for the
// category 'iCat'. Return 0 if the category doesn't exists
#if BUILDMODE != 0
inline
#endif
long _GDSGetSizeCat(const GDataSet* const that, const long iCat) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (that->_split == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that->_split' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (iCat < 0 || iCat >= GDSGetNbCat(that)) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'iCat' is invalid (0<=%ld<%ld)",
      iCat, GDSGetNbCat(that));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return VecGet(that->_split, iCat);
}

// Unsplit the GDataSet 'that', i.e. after calling GDataSetUnsplit 'that'
// has only one category containing all the samples
#if BUILDMODE != 0
inline
#endif
void _GDSUnsplit(GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Unsplitting is equivalent to splitting in one category with all the
```

```
  // samples
  VecShort* split = VecShortCreate(1);
  VecSet(split, 0, GDSGetSize(that));
  GDSSplit(that, split);
  VecFree(&split);
}

// Shuffle the samples of the category 'iCat' of the GDataSet 'that'.
// Reset the iterator of the category
#if BUILDMODE != 0
inline
#endif
void _GDSShuffle(GDataSet* const that, const long iCat) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (that->_categories == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that->_categories' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (iCat < 0 || iCat >= GDSGetNbCat(that)) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'iCat' is invalid (0<=%ld<%ld)",
      iCat, GDSGetNbCat(that));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Shuffle the GSet of the category
  GSetShuffle(that->_categories + iCat);
  // Reset the iterator
  GDSReset(that, iCat);
}

// Shuffle the samples of all the categories of the GDataSet 'that'.
// Reset the iterator of the categories
#if BUILDMODE != 0
inline
#endif
void _GDSShuffleAll(GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Loop on categories
  for (int iCat = GDSGetNbCat(that); iCat--;)
    // Shuffle the category
    GDSShuffle(that, iCat);
}

// Get the name of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const char* _GDSName(const GDataSet* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return that->_name;
}

// Get the description of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const char* _GDSDesc(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return that->_desc;
}

// Get the path of the config file of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const char* _GDSCfgFilePath(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return that->_cfgFilePath;
}

// Get a copy of the path of the config file of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
char* _GDSGetCfgFilePath(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  char* ret = malloc(strlen(that->_cfgFilePath) + 1);
  strcpy(ret, that->_cfgFilePath);
  return ret;
}

// Get the path of the folder of the config file of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
char* _GDSGetCfgFolderPath(const GDataSet* const that) {
#if BUILDMODE == 0
```

```
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return PBFSGetRootPath(that->_cfgFilePath);
}

// Get the type of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
GDataSetType _GDSGetType(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return that->_type;
}

// Get the number of categories of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
long _GDSGetNbCat(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  return VecGetDim(that->_split);
}

// If there is a next sample move to the next sample of the category
// 'iCat' and return true, else return false
#if BUILDMODE != 0
inline
#endif
bool _GDSStepSample(const GDataSet* const that, const long iCat) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (that->_iterators == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that->_iterators' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (iCat < 0 || iCat >= GDSGetNbCat(that)) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'iCat' is invalid (0<=%ld<%ld)",
      iCat, GDSGetNbCat(that));
    PBErrCatch(PBImgAnalysisErr);
  }
```

```
#endif
  return GSetIterStep(that->_iterators + iCat);
}

// Reset the iterator on category 'iCat' of the GDataSet 'that', i.e.
// the next call to GDataSetGetNextSample will give the first sample of
// the category 'iCat'
#if BUILDMODE != 0
inline
#endif
void _GDSReset(GDataSet* const that, const long iCat) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (that->_iterators == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that->_iterators' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
  if (iCat < 0 || iCat >= GDSGetNbCat(that)) {
    GDataSetErr->_type = PBErrTypeInvalidArg;
    sprintf(PBImgAnalysisErr->_msg, "'iCat' is invalid (0<=%ld<%ld)",
      iCat, GDSGetNbCat(that));
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  GSetIterReset(that->_iterators + iCat);
}

// Reset the iterator on all categories of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
void _GDSResetAll(GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
  // Loop on categories
  for (int iCat = GDSGetNbCat(that); iCat--;)
    // Shuffle the category
    GDSReset(that, iCat);
}

// Get the dimensions of the samples of GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const VecShort* _GDSSampleDim(const GDataSet* const that) {
#if BUILDMODE == 0
  if (that == NULL) {
    GDataSetErr->_type = PBErrTypeNullPointer;
    sprintf(PBImgAnalysisErr->_msg, "'that' is null");
    PBErrCatch(PBImgAnalysisErr);
  }
#endif
```

```
  return that->_sampleDim;
}
```

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=gdataset
$($(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 Dataset configuration file

4.1 VecFloat

```
{
  "dataSet": "testGDataSet",
  "dataSetType": "0",
  "desc": "UnitTestGDataSetCreateFree",
  "dim": {
    "_dim":"1",
    "_val":["2"]
  },
  "nbSample": "3",
  "samples": [
    {
```

```
      "_dim":"2",
      "_val":["0.0","1.0"]
    },
    {
      "_dim":"2",
      "_val":["2.0","3.0"]
    },
    {
      "_dim":"2",
      "_val":["4.0","5.0"]
    }
  ]
}
```

4.2   Pair of GenBrush

```
{
  "dataSet": "testGDataSetGenBrushPair",
  "dataSetType": "1",
  "desc": "UnitTestGDataSetGenBrushPair",
  "dim": {
    "_dim":"2",
    "_val":["20", "10"]
  },
  "nbSample": "2",
  "samples": [
    {
      "img":"img000.tga",
      "mask":"mask000.tga"
    },
    {
      "img":"img001.tga",
      "mask":"mask001.tga"
    }
  ]
}
```

5   Unit tests

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

void UnitTestGDataSetVecFloatCreateFree() {
  srandom(1);
  char* cfgFilePath = "testGDataSetVecFloat.json";
  GDataSetVecFloat gdataset = GDataSetVecFloatCreateStatic(cfgFilePath);
  if (strcmp(gdataset._dataSet._cfgFilePath, cfgFilePath) != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDataSetCreateStatic failed");
    PBErrCatch(GDataSetErr);
  }
  GDataSet* g = (GDataSet*)(&gdataset);
  if (GSetGet(g->_categories, 0) != GSetGet(&(g->_samples), 0) ||
```

```
      GSetGet(g->_categories, 1) != GSetGet(&(g->_samples), 1) ||
      GSetGet(g->_categories, 2) != GSetGet(&(g->_samples), 2)) {
      GDataSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GDataSetErr->_msg, "GDataSetCreateStatic failed");
      PBErrCatch(GDataSetErr);
    }
  GDataSetVecFloatFreeStatic(&gdataset);
  printf("UnitTestGDataSetVecFloatCreateFree OK\n");
}

void UnitTestGDataSetVecFloatGet() {
  srandom(1);
  char* cfgFilePath = "testGDataSetVecFloat.json";
  GDataSetVecFloat gdataset = GDataSetVecFloatCreateStatic(cfgFilePath);
  if (strcmp(GDSCfgFilePath(&gdataset), cfgFilePath) != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSCfgFilePath failed");
    PBErrCatch(GDataSetErr);
  }
  char* str = GDSGetCfgFilePath(&gdataset);
  if (strcmp(str, cfgFilePath) != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetCfgFilePath failed");
    PBErrCatch(GDataSetErr);
  }
  free(str);
  str = GDSGetCfgFolderPath(&gdataset);
  if (strcmp(str, "") != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetCfgFolderPath failed");
    PBErrCatch(GDataSetErr);
  }
  free(str);
  if (strcmp(GDSDesc(&gdataset), "UnitTestGDataSetCreateFree") != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSDesc failed");
    PBErrCatch(GDataSetErr);
  }
  if (strcmp(GDSName(&gdataset), "testGDataSet") != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSName failed");
    PBErrCatch(GDataSetErr);
  }
  if (GDSGetType(&gdataset) != GDataSetType_VecFloat) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetType failed");
    PBErrCatch(GDataSetErr);
  }
  if (GDSGetNbCat(&gdataset) != 1) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetNbCat failed");
    PBErrCatch(GDataSetErr);
  }
  if (GDSGetSize(&gdataset) != 3) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetSize failed");
    PBErrCatch(GDataSetErr);
  }
  if (GDSGetSizeCat(&gdataset, 0) != 3) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetSizeCat failed");
    PBErrCatch(GDataSetErr);
```

24

```
  }
  VecShort* dim = VecShortCreate(1);
  VecSet(dim, 0, 2);
  if (VecIsEqual(GDSSampleDim(&gdataset), dim) != true) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSSampleDim failed");
    PBErrCatch(GDataSetErr);
  }
  VecFree(&dim);
  GDataSetVecFloatFreeStatic(&gdataset);
  printf("UnitTestGDataSetVecFloatGet OK\n");
}

void UnitTestGDataSetVecFloatSplitUnsplit() {
  srandom(1);
  char* cfgFilePath = "testGDataSetVecFloat.json";
  GDataSetVecFloat gdataset = GDataSetVecFloatCreateStatic(cfgFilePath);
  VecShort* split = VecShortCreate(2);
  VecSet(split, 0, 1);
  VecSet(split, 1, 2);
  GDSSplit(&gdataset, split);
  if (GDSGetNbCat(&gdataset) != 2) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSSplit failed");
    PBErrCatch(GDataSetErr);
  }
  if (GDSGetSizeCat(&gdataset, 0) != 1 ||
    GDSGetSizeCat(&gdataset, 1) != 2) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSSplit failed");
    PBErrCatch(GDataSetErr);
  }
  GDSUnsplit(&gdataset);
  if (GDSGetNbCat(&gdataset) != 1) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSUnsplit failed");
    PBErrCatch(GDataSetErr);
  }
  VecFree(&split);
  GDataSetVecFloatFreeStatic(&gdataset);
  printf("UnitTestGDataSetVecFloatSplitUnsplit OK\n");
}

void UnitTestGDataSetVecFloatShuffle() {
  srandom(1);
  char* cfgFilePath = "testGDataSetVecFloat.json";
  GDataSetVecFloat gdataset = GDataSetVecFloatCreateStatic(cfgFilePath);
  GDSShuffle(&gdataset, 0);
  GDataSet* g = (GDataSet*)(&gdataset);
  if (GSetGet(g->_categories, 0) != GSetGet(&(g->_samples), 1)/* ||
    GSetGet(g->_categories, 1) != GSetGet(&(g->_samples), 0) ||
    GSetGet(g->_categories, 2) != GSetGet(&(g->_samples), 1)*/) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSShuffle failed");
    PBErrCatch(GDataSetErr);
  }
  GDataSetVecFloatFreeStatic(&gdataset);
  printf("UnitTestGDataSetVecFloatShuffle OK\n");
}

void UnitTestGDataSetVecFloatStepSampleGetSample() {
  srandom(1);
```

```c
  char* cfgFilePath = "testGDataSetVecFloat.json";
  GDataSetVecFloat gdataset = GDataSetVecFloatCreateStatic(cfgFilePath);
  int iSample = 0;
  float check[6] = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0};
  do {
    VecFloat* sample = GDSGetSample(&gdataset, 0);
    if (ISEQUALF(VecGet(sample, 0), check[iSample * 2]) == false ||
      ISEQUALF(VecGet(sample, 1), check[iSample * 2 + 1]) == false) {
      GDataSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GDataSetErr->_msg, "GDSGetSample failed");
      PBErrCatch(GDataSetErr);
    }
    VecFree(&sample);
    ++iSample;
  } while (GDSStepSample(&gdataset, 0));
  GDataSetVecFloatFreeStatic(&gdataset);
  printf("UnitTestGDataSetVecFloatStepSampleGetSample OK\n");
}

void UnitTestGDataSetVecFloat() {
  UnitTestGDataSetVecFloatCreateFree();
  UnitTestGDataSetVecFloatGet();
  UnitTestGDataSetVecFloatSplitUnsplit();
  UnitTestGDataSetVecFloatShuffle();
  UnitTestGDataSetVecFloatStepSampleGetSample();
}

void UnitTestGDataSetGenBrushPair() {
  srandom(1);
  char* cfgFilePath = "testGDataSetGenBrushPair.json";
  GDataSetGenBrushPair gdataset =
    GDataSetGenBrushPairCreateStatic(cfgFilePath);
  do {
    GDSGenBrushPair* sample = GDSGetSample(&gdataset, 0);
    if (VecIsEqual(GBDim(sample->_img),
      GDSSampleDim(&gdataset)) == false ||
      VecIsEqual(GBDim(sample->_mask),
      GDSSampleDim(&gdataset)) == false) {
      GDataSetErr->_type = PBErrTypeUnitTestFailed;
      sprintf(GDataSetErr->_msg, "GDSGetSample<GenBrushPair> failed");
      PBErrCatch(GDataSetErr);
    }
    GDSGenBrushPairFree(&sample);
  } while (GDSStepSample(&gdataset, 0));
  GDataSetGenBrushPairFreeStatic(&gdataset);
  printf("UnitTestGDataSetGenBrushPair OK\n");
}

void UnitTestAll() {
  UnitTestGDataSetVecFloat();
  UnitTestGDataSetGenBrushPair();
}

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

6 Unit test output

```
UnitTestGDataSetVecFloatCreateFree OK
UnitTestGDataSetVecFloatCreateFree OK
UnitTestGDataSetVecFloatSplitUnsplit OK
UnitTestGDataSetVecFloatShuffle OK
UnitTestGDataSetVecFloatStepSampleGetSample OK
UnitTestGDataSetGenBrushPair OK
```