

# GDataSet

P. Baillehache

April 21, 2019

## Contents

<b>1</b>	<b>Interface</b>	<b>2</b>
<b>2</b>	<b>Code</b>	<b>9</b>
2.1	gdataset.c . . . . .	9
2.2	gdataset-inline.c . . . . .	20
<b>3</b>	<b>Makefile</b>	<b>25</b>
<b>4</b>	<b>Dataset configuration file</b>	<b>26</b>
4.1	VecFloat . . . . .	26
4.2	Pair of GenBrush . . . . .	26
<b>5</b>	<b>Unit tests</b>	<b>27</b>
<b>6</b>	<b>Unit test output</b>	<b>32</b>

## 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 include 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;
    // Nb of mask per img
    int _nbMask;
} GDataSetGenBrushPair;

#define GDS_NBMAXMASK 100
typedef struct GDSFilePathPair {
    char* _path[1 + GDS_NBMAXMASK];
} GDSFilePathPair;

typedef struct GDSGenBrushPair {
    GenBrush* _img;
    GenBrush* _mask[GDS_NBMAXMASK];
} GDSGenBrushPair;

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

// Create a new GDataSet defined by the file at 'cfgFilePath'
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'
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'
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'
#ifdef BUILDMODE != 0
inline
#endif
long _GDSGetSize(const GDataSet* const that);

// Get the number of masks in the GDataSet 'that'
int _GDSGetNbMask(const GDataSet* const that);

// Get the number of masks in the GDataSetGenBrushPair 'that'
#ifdef BUILDMODE != 0
inline
#endif
int GDSGetNbMaskGenBrushPair(const GDataSetGenBrushPair* const that);

// Get the total number of samples in the GDataSet 'that' for the
// category 'iCat'. Return 0 if the category doesn't exists
#ifdef 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 be 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
#ifdef 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
#ifdef 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
#ifdef 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);
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);

// Get the samples of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const GSetVecFloat* _GDSVecFloatSamples(
    const GDataSetVecFloat* const that);
#if BUILDMODE != 0
inline
#endif
const GSet* _GDSGenBrushPairSamples(
    const GDataSetGenBrushPair* const that);

// Center the GDataSet 'that' on its mean
void GDSMeanCenter(GDataSetVecFloat* const that);

// Get the mean of the GDataSet 'that'
VecFloat* GDSGetMean(const GDataSetVecFloat* const that);

// Get a clone of the GDataSet 'that'
// All the data in the GDataSet are cloned except for the splitting
// categories which are reset to one category made of the original data
GDataSetVecFloat GDSClone(const GDataSetVecFloat* 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)

```

```

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 GDSGetNbMask(DataSet) _Generic(DataSet, \
    GDataSet*: _GDSGetNbMask, \
    const GDataSet*: _GDSGetNbMask, \
    GDataSetGenBrushPair*: GDSGetNbMaskGenBrushPair, \
    const GDataSetGenBrushPair*: GDSGetNbMaskGenBrushPair, \
    default: PBErrInvalidPolymorphism)(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 GDSSStepSample(DataSet, ICat) _Generic(DataSet, \
    GDataSet*: _GDSSStepSample, \
    const GDataSet*: _GDSSStepSample, \
    GDataSetVecFloat*: _GDSSStepSample, \
    const GDataSetVecFloat*: _GDSSStepSample, \
    GDataSetGenBrushPair*: _GDSSStepSample, \
    const GDataSetGenBrushPair*: _GDSSStepSample, \
    default: PBErrInvalidPolymorphism)((const GDataSet*)DataSet, ICat)

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

#define GDSSamples(DataSet) _Generic(DataSet, \
    GDataSetVecFloat*: _GDSVecFloatSamples, \
    const GDataSetVecFloat*: _GDSVecFloatSamples, \
    GDataSetGenBrushPair*: _GDSGenBrushPairSamples, \
    const GDataSetGenBrushPair*: _GDSGenBrushPairSamples, \
    default: PBErrInvalidPolymorphism)(DataSet)

// ===== Inline =====

#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'
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));
// 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);
    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'
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);
    }
    // Get the nb of mask
    JSONNode* prop = JSONProperty(that._dataSet._json, "nbMask");
    if (prop == NULL) {
        GDataSetErr->_type = PBErrTypeInvalidData;
        sprintf(GDataSetErr->_msg,
            "Invalid description file (nbMask missing)");
        PBErrCatch(GDataSetErr);
    }
    that._nbMask = atoi(JSONLabel(JSONValue(prop), 0));
    if (that._nbMask >= GDS_NBMAXMASK) {
        GDataSetErr->_type = PBErrTypeInvalidData;
        sprintf(GDataSetErr->_msg,
            "Invalid description file (invalid nbMask %d>=%d)",
            that._nbMask, GDS_NBMAXMASK);
        PBErrCatch(GDataSetErr);
    }
    // Load the samples
    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));
pair->_path[0] = NULL;
for (int iMask = GDS_NBMAXMASK; iMask--;)
    pair->_path[1 + iMask] = NULL;
// 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);
}
for (int iMask = 0; iMask < that._nbMask; ++iMask) {
    subVal = JSONValue(subProp, iMask);
    pair->_path[1 + iMask] = PBErrMalloc(GDataSetErr,
        sizeof(char) * (strlen(JSONLabel(subVal)) + 1));
    strcpy(pair->_path[1 + iMask], 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 be 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) {
#ifdef 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
    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;
    }
}

// Get the number of masks in the GDataSet 'that'
int _GDSGetNbMask(const GDataSet* const that) {
    // Call the appropriate function according to the type
    switch (GDSGetType(that)) {
        case GDataSetType_GenBrushPair:
            return GDSGetNbMaskGenBrushPair((GDataSetGenBrushPair*)that);
            break;
        default:
            return 0;
            break;
    }
}

VecFloat* GDSGetSampleVecFloat(
    const GDataSetVecFloat* const that, const int iCat) {
#ifdef 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) {
#ifdef 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
    GDSFilePathPair* pairFile =
        GSetIterGet(((GDataSet*)that)->_iterators + iCat);
    GDSGenBrushPair* pairSample = PBErrMalloc(GDataSetErr,
        sizeof(GDSGenBrushPair));
    for (int iMask = 0; iMask < GDS_NBMAXMASK; ++iMask)
        pairSample->_mask[iMask] = NULL;
    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 (gb != NULL && !VecIsEqual(GBDim(gb), GDSSampleDim(that))) {
        pairSample->_img = GBScale(gb,
            (const VecShort2D*)GDSSampleDim(that), GBScaleMethod_Default);
        GBFree(&gb);
    } else {
        pairSample->_img = gb;
    }
    free(path);
    for (int iMask = 0; iMask < GDSGetNbMask(that); ++iMask) {
        path = PBFSJoinPath(root, pairFile->_path[1 + iMask]);
        gb = GBCreateFromFile(path);
        if (gb != NULL && !VecIsEqual(GBDim(gb), GDSSampleDim(that))) {
            pairSample->_mask[iMask] = GBScale(gb,
                (const VecShort2D*)GDSSampleDim(that), GBScaleMethod_Default);
            GBFree(&gb);
        } else {
            pairSample->_mask[iMask] = 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;
    for (int iMask = GDS_NBMAXMASK + 1; iMask--;)
        if ((*that)->_path[iMask] != NULL)
            free((*that)->_path[iMask]);
    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));
    for (int iMask = GDS_NBMAXMASK; iMask--;)
        GBFree(&((*that)->_mask[iMask]));
    free(*that);
    *that = NULL;
}

// Center the GDataSet 'that' on its mean
void GDSMeanCenter(GDataSetVecFloat* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GDataSetErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    // Get the mean of the dataset
    VecFloat* mean = GDSGetMean(that);
    // Translate all the data by the mean of the data set
    if (GDSGetSize(that) > 0) {
        GSetIterForward iter = GSetIterForwardCreateStatic(GDSSamples(that));
        do {
            VecFloat* sample = GSetIterGet(&iter);
            VecOp(sample, 1.0, mean, -1.0);
        } while (GSetIterStep(&iter));
    }
    // Free memory
    VecFree(&mean);
}

// Get the mean of the GDataSet 'that'
VecFloat* GDSGetMean(const GDataSetVecFloat* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GDataSetErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    // Get the dimension of the samples
    const VecShort* dim = GDSSampleDim(that);
    // Create a vector to calculate the mean
    VecFloat* mean = VecFloatCreate(VecGet(dim, 0));
    // Calculate the mean
    if (GDSGetSize(that) > 0) {
        GSetIterForward iter =
            GSetIterForwardCreateStatic(GDSSamples(that));
        do {
            VecFloat* v = GSetIterGet(&iter);
            VecOp(mean, 1.0, v, 1.0);
        } while (GSetIterStep(&iter));
        VecScale(mean, 1.0 / (float)GDSGetSize(that));
    }
    // Return the result
    return mean;
}

```

```

// Get a clone of the GDataSet 'that'
// All the data in the GDataSet are cloned except for the splitting
// categories which are reset to one category made of the original data
GDataSetVecFloat GDSClone(const GDataSetVecFloat* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        GDataSetErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    // Declare the result dataset
    GDataSetVecFloat dataset;
    // Create a pointer to the GDataSet for convenience
    GDataSet* tho = &(dataset._dataSet);
    // Clone or initialize the properties
    tho->_json = NULL;
    tho->_cfgFilePath = PBErrMalloc(GDataSetErr,
        sizeof(char) * (1 + strlen(that->_dataSet._cfgFilePath)));
    strcpy(tho->_cfgFilePath, that->_dataSet._cfgFilePath);
    tho->_name = PBErrMalloc(GDataSetErr,
        sizeof(char) * (1 + strlen(that->_dataSet._name)));
    strcpy(tho->_name, that->_dataSet._name);
    tho->_desc = PBErrMalloc(GDataSetErr,
        sizeof(char) * (1 + strlen(that->_dataSet._desc)));
    strcpy(tho->_desc, that->_dataSet._desc);
    tho->_type = that->_dataSet._type;
    tho->_nbSample = that->_dataSet._nbSample;
    tho->_sampleDim = VecClone(that->_dataSet._sampleDim);
    tho->_samples = GSetCreateStatic();
    if (GDSSGetSize(that) > 0) {
        GSetIterForward iter = GSetIterForwardCreateStatic(GDSSSamples(that));
        do {
            VecFloat* v = GSetIterGet(&iter);
            GSetAppend(&(tho->_samples), VecClone(v));
        } while (GSetIterStep(&iter));
    }
    tho->_split = NULL;
    tho->_categories = NULL;
    tho->_iterators = NULL;
    tho->_split = VecShortCreate(1);
    VecSet(tho->_split, 0, tho->_nbSample);
    tho->_categories = PBErrMalloc(GDataSetErr, sizeof(GSet));
    tho->_categories[0] = GSetCreateStatic();
    if (GDSSGetSize(that) > 0) {
        GSetIterForward iter =
            GSetIterForwardCreateStatic(&(tho->_samples));
        do {
            void* sample = GSetIterGet(&iter);
            GSetAppend(tho->_categories, sample);
        } while (GSetIterStep(&iter));
    }
    tho->_iterators =
        PBErrMalloc(GDataSetErr, sizeof(GSetIterForward));
    tho->_iterators[0] =
        GSetIterForwardCreateStatic(tho->_categories);
    // Return the result dataset
    return dataset;
}

```

## 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
}

```

```

#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;
}

// Get the number of masks in the GDataSetGenBrushPair 'that'
#if BUILDMODE != 0
inline
#endif
int GDSGetNbMaskGenBrushPair(const GDataSetGenBrushPair* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GDataSetErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    return that->_nbMask;
}

// Get the samples of the GDataSet 'that'
#if BUILDMODE != 0
inline
#endif
const GSetVecFloat* _GDSVecFloatSamples(
    const GDataSetVecFloat* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GDataSetErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    return (GSetVecFloat*)&(that->_dataSet._samples);
}

#if BUILDMODE != 0
inline
#endif
const GSet* _GDSGenBrushPairSamples(
    const GDataSetGenBrushPair* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        GDataSetErr->_type = PBErrTypeNullPointer;
        sprintf(PBImgAnalysisErr->_msg, "'that' is null");
        PBErrCatch(PBImgAnalysisErr);
    }
#endif
    return &(that->_dataSet._samples);
}

```

### 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": "dataset-002-001",

```

```

"datasetType": "1",
"desc": "unitTest",
"dim": {
  "_dim": "2",
  "_val": [
    "10",
    "20"
  ]
},
"format": "tga",
"nbMask": "2",
"nbSample": "3",
"samples": [
  {
    "bounding": [],
    "img": "img000.tga",
    "mask": [
      "mask000-000.tga",
      "mask000-001.tga"
    ]
  },
  {
    "bounding": [],
    "img": "img001.tga",
    "mask": [
      "mask001-000.tga",
      "mask001-001.tga"
    ]
  },
  {
    "bounding": [],
    "img": "img002.tga",
    "mask": [
      "mask002-000.tga",
      "mask002-001.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 UnitTestGDataSetVecFloatCreateFreeClone() {
  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);
}
GDataSetVecFloat clone = GDSClone(&gdataset);
if (strcmp(clone._dataSet._cfgFilePath, cfgFilePath) != 0) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSClone failed");
    PBErrCatch(GDataSetErr);
}
GDataSet* f = (GDataSet*)&clone;
if (GSetGet(f->_categories, 0) != GSetGet(&(f->_samples), 0) ||
    GSetGet(f->_categories, 1) != GSetGet(&(f->_samples), 1) ||
    GSetGet(f->_categories, 2) != GSetGet(&(f->_samples), 2)) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSClone failed");
    PBErrCatch(GDataSetErr);
}

GDataSetVecFloatFreeStatic(&clone);
GDataSetVecFloatFreeStatic(&gdataset);
printf("UnitTestGDataSetVecFloatCreateFreeClone 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);
}
if ((GSet*)GDSSamples(&gdataset) != &(gdataset._dataSet._samples)) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSSamples failed");
    PBErrCatch(GDataSetErr);
}
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);
VecFloat* mean = GDSGetMean(&gdataset);
VecFloat2D checkMean = VecFloatCreateStatic2D();
VecSet(&checkMean, 0, 2.0);
VecSet(&checkMean, 1, 3.0);
if (!VecIsEqual(mean, &checkMean)) {
    GDataSetErr->_type = PBErrTypeUnitTestFailed;
    sprintf(GDataSetErr->_msg, "GDSGetMean failed");
    PBErrCatch(GDataSetErr);
}
VecFree(&mean);
GDSSampleCenter(&gdataset);
VecFloat2D checkMeanCenter[3];
for (int i = 0; i < GDSGetSize(&gdataset); ++i) {
    checkMeanCenter[i] = VecFloatCreateStatic2D();
    VecSet(checkMeanCenter + i, 0, -2.0 + (float)i * 2.0);
    VecSet(checkMeanCenter + i, 1, -2.0 + (float)i * 2.0);
}
GSetIterForward iter = GSetIterForwardCreateStatic(
    GDSSamples(&gdataset));
int i = 0;
do {
    VecFloat* sample = GSetIterGet(&iter);
    if (!VecIsEqual(sample, checkMeanCenter + i)) {
        GDataSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg, "GDSMeanCenter failed");
        PBErrCatch(GDataSetErr);
    }
} while (GSetIterStep(&iter) && ++i);

GDataSetVecFloatFreeStatic(&gdataset);

```

```

    printf("UnitTestDataSetVecFloatGet OK\n");
}

void UnitTestDataSetVecFloatSplitUnsplit() {
    srand(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 = PErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg, "GDSSplit failed");
        PErrCatch(GDataSetErr);
    }
    if (GDSGetSizeCat(&gdataset, 0) != 1 ||
        GDSGetSizeCat(&gdataset, 1) != 2) {
        GDataSetErr->_type = PErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg, "GDSSplit failed");
        PErrCatch(GDataSetErr);
    }
    GDSUnsplit(&gdataset);
    if (GDSGetNbCat(&gdataset) != 1) {
        GDataSetErr->_type = PErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg, "GDSUnsplit failed");
        PErrCatch(GDataSetErr);
    }
    VecFree(&split);
    GDataSetVecFloatFreeStatic(&gdataset);
    printf("UnitTestDataSetVecFloatSplitUnsplit OK\n");
}

void UnitTestDataSetVecFloatShuffle() {
    srand(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 = PErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg, "GDSShuffle failed");
        PErrCatch(GDataSetErr);
    }
    GDataSetVecFloatFreeStatic(&gdataset);
    printf("UnitTestDataSetVecFloatShuffle OK\n");
}

void UnitTestDataSetVecFloatStepSampleGetSample() {
    srand(1);
    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 = PErrTypeUnitTestFailed;
            sprintf(GDataSetErr->_msg, "GDSGetSample failed");
        }
    }

```

```

        PBErrCatch(GDataSetErr);
    }
    VecFree(&sample);
    ++iSample;
} while (GDSSStepSample(&gdataset, 0));
GDataSetVecFloatFreeStatic(&gdataset);
printf("UnitTestGDataSetVecFloatStepSampleGetSample OK\n");
}

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

void UnitTestGDataSetGenBrushPair() {
    srand(1);
    char* cfgFilePath = "testGDataSetGenBrushPair.json";
    GDataSetGenBrushPair gdataset =
        GDataSetGenBrushPairCreateStatic(cfgFilePath);
    if (GDSGetNbMask(&gdataset) != 2) {
        GDataSetErr->_type = PBErrTypeUnitTestFailed;
        sprintf(GDataSetErr->_msg, "GDSGetSample<GenBrushPair> failed");
        PBErrCatch(GDataSetErr);
    }
    int iCat = 0;
    do {
        GDSGenBrushPair* sample = GDSGetSample(&gdataset, iCat);
        if (VecIsEqual(GBDim(sample->_img),
            GDSSampleDim(&gdataset)) == false ||
            VecIsEqual(GBDim(sample->_mask[0]),
            GDSSampleDim(&gdataset)) == false ||
            VecIsEqual(GBDim(sample->_mask[1]),
            GDSSampleDim(&gdataset)) == false) {
            GDataSetErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GDataSetErr->_msg, "GDSGetSample<GenBrushPair> failed");
            PBErrCatch(GDataSetErr);
        }
        GDSGenBrushPairFree(&sample);
    } while (GDSSStepSample(&gdataset, iCat));
    GDataSetGenBrushPairFreeStatic(&gdataset);
    printf("UnitTestGDataSetGenBrushPair OK\n");
}

void UnitTestSDSIA() {
    srand(1);
    char* cfgFilePath = "../SDSIA/UnitTestOut/002/001/dataset.json";
    GDataSetGenBrushPair gdataset =
        GDataSetGenBrushPairCreateStatic(cfgFilePath);
    int iCat = 0;
    do {
        GDSGenBrushPair* sample = GDSGetSample(&gdataset, iCat);
        if (VecIsEqual(GBDim(sample->_img),
            GDSSampleDim(&gdataset)) == false ||
            VecIsEqual(GBDim(sample->_mask[0]),
            GDSSampleDim(&gdataset)) == false ||
            VecIsEqual(GBDim(sample->_mask[1]),
            GDSSampleDim(&gdataset)) == false) {
            GDataSetErr->_type = PBErrTypeUnitTestFailed;
            sprintf(GDataSetErr->_msg, "GDSGetSample<GenBrushPair> failed");

```

```

        PBErrCatch(GDataSetErr);
    }
    GDSEnBrushPairFree(&sample);
} while (GDSStepSample(&gdataset, iCat));
GDataSetGenBrushPairFreeStatic(&gdataset);
printf("UnitTestSDSIA OK\n");
}

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

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

```

## 6 Unit test output

```

UnitTestGDataSetVecFloatCreateFreeClone OK
UnitTestGDataSetVecFloatGet OK
UnitTestGDataSetVecFloatSplitUnsplit OK
UnitTestGDataSetVecFloatShuffle OK
UnitTestGDataSetVecFloatStepSampleGetSample OK
UnitTestGDataSetGenBrushPair OK
UnitTestSDSIA OK

```