

# BCurve

P. Baillehache

September 9, 2020

## Contents

<b>1</b>	<b>Definitions</b>	<b>2</b>
1.1	BCurve definition . . . . .	2
1.2	BCurve from cloud points . . . . .	3
1.3	BBody definition . . . . .	3
<b>2</b>	<b>Interface</b>	<b>4</b>
<b>3</b>	<b>Code</b>	<b>20</b>
3.1	bcurve.c . . . . .	20
3.2	bcurve-inline.c . . . . .	55
<b>4</b>	<b>Makefile</b>	<b>91</b>
<b>5</b>	<b>Unit tests</b>	<b>91</b>
<b>6</b>	<b>Unit tests output</b>	<b>128</b>

## Introduction

BCurve is C library to manipulate geometry based on Bezier curves of any dimension and order.

It offers function to create, clone, load and save (JSON format), and modify a curve or surface, to print it, to scale, rotate (in 2D) or translate it, to get the weights (coefficients of each control point given the value of the parameter of the curve), and to get the bounding box.

BCurve objects are Bezier curves from 1D to ND. For BCurve object, the library offers functions to get its approximate length (sum of distance between control points), and to create a BCurve connecting points of a point cloud.

SCurve objects are a set of BCurve (called segments) continuously connected and has the same interface as a BCurve, plus function to add and remove segments, and apply the Chaikin subdivision algorithm on curve of order 1.

BBody objects are extension of BCurve objects for the case M dimensions to N dimensions. If M equals 1 it is equivalent to a BCurve. If M equals 2 it is equivalent to a surface in N dimension. If M equals 3 it is equivalent to a volume. Note that by using one dimension as the time dimension one can describes the movement of a curve, surface, etc... over time. The library offers the same functions for a BBody as for a BCurve.

It uses the PBErr, PBMath, GSet, Shapoid libraries.

# 1 Definitions

## 1.1 BCurve definition

A BCurve  $B$  is defined by its dimension  $D \in \mathbb{N}_+^*$ , its order  $O \in \mathbb{N}_+$  and its  $(O + 1)$  control points  $\vec{C}_i \in \mathbb{R}^D$ . The curve in dimension  $D$  associated to the BCurve  $B$  is defined by  $\vec{B}(t)$ :

$$\begin{cases} \vec{B}(t) = \sum_{i=0}^O W_i^O(t) \vec{C}_i & \text{if } t \in [0.0, 1.0] \\ \vec{B}(t) = \vec{C}_0 & \text{if } t < 0.0 \\ \vec{B}(t) = \vec{C}_O & \text{if } t > 1.0 \end{cases} \quad (1)$$

where, if  $O = 0$

$$W_0^0(t) = 1.0 \quad (2)$$

and if  $O \neq 0$

$$\begin{cases} W_0^1(t) = 1.0 - t \\ W_1^1(t) = t \\ W_{-1}^i(t) = 0.0 \\ W_j^i(t) = (1.0 - t)W_j^{i-1}(t) + tW_{j-1}^{i-1}(t) \text{ for } i \in [2, O], j \in [0, i] \end{cases} \quad (3)$$

## 1.2 BCurve from cloud points

Given the cloud points made of  $N$  points  $\vec{P}_i$ , the BCurve of order  $N - 1$  passing through the  $N$  points (in the same order  $\vec{P}_0, \vec{P}_1, \vec{P}_2, \dots$  as given in input) can be obtained as follow.

If  $N = 1$  the solution is trivial:  $\vec{C}_0 = \vec{P}_0$ . As well, if  $N = 2$  the solution is trivial:  $\vec{C}_0 = \vec{P}_0$  and  $\vec{C}_1 = \vec{P}_1$ .

If  $N > 2$ , we need first to define the  $N$  values  $t_i$  corresponding to each  $\vec{P}_i$  ( $\vec{B}(t_i) = \vec{P}_i$ ). We will consider here  $t_i$  such as

$$t_i = \frac{L(\vec{P}_i)}{L(\vec{P}_{N-1})} \quad (4)$$

where

$$\begin{cases} L(P_0) = 0.0 \\ L(P_i) = \sum_{j=1}^i \left\| \overrightarrow{P_{j-1}P_j} \right\| \end{cases} \quad (5)$$

then we can calculate the  $C_i$  as follow. We have  $\vec{C}_0 = \vec{P}_0$  and  $\vec{C}_{N-1} = \vec{P}_{N-1}$ , and others  $\vec{C}_i$  can be obtained by solving the linear system below for each dimension:

$$\begin{bmatrix} W_1^{N-1}(t_1) & \dots & W_{N-2}^{N-1}(t_1) \\ \vdots & \ddots & \vdots \\ W_1^{N-1}(t_{N-2}) & \dots & W_{N-2}^{N-1}(t_{N-2}) \end{bmatrix} \begin{bmatrix} C_1 \\ \vdots \\ C_{N-2} \end{bmatrix} = \begin{bmatrix} P_1 - (W_0^{N-1}(t_1)P_0 + W_{N-1}^{N-1}(t_1)P_{N-1}) \\ \vdots \\ P_{N-2} - (W_0^{N-1}(t_{N-2})P_0 + W_{N-1}^{N-1}(t_{N-2})P_{N-1}) \end{bmatrix} \quad (6)$$

## 1.3 BBody definition

A BBody  $A$  is defined by its input dimension  $D_i \in \mathbb{N}_+^*$ , its output dimension  $D_o \in \mathbb{N}_+^*$ , its order  $O \in \mathbb{N}_+$  and its  $(O + 1)^{D_i}$  control points  $\vec{C}_i \in \mathbb{R}^{D_o}$ . Con-

trol points indices are ordered as follow (for an example BBody with  $D_i = 3$ ):  
 $(0,0,0), (0,0,1), \dots, (0,0,O+1), (0,1,0), (0,1,1), \dots$

Note that if  $D_i$  is equal to 1, a BBody is equivalent to a BCurve.

The function  $\vec{A}() : [0.0, 1.0]^{D_i} \mapsto \mathbb{R}^{D_o}$  associated to the BBody  $A$  is defined by:

$$\vec{A}(\vec{u}) = \vec{R}_A(\vec{0}, \vec{u}, 0) \quad (7)$$

where

$$\begin{cases} \vec{R}_A(\vec{c}, \vec{u}, d) = \overrightarrow{B_{\{\vec{C}_{I(\vec{c}, d)}\}}}(u_d) & \text{if } d = D_i - 1 \\ \vec{R}_A(\vec{c}, \vec{u}, d) = \overrightarrow{B_{\{\vec{R}_S(\{\vec{c}\}_d, \vec{u}, d+1)\}}}(u_d) & \text{if } d \neq D_i - 1 \end{cases} \quad (8)$$

where  $\overrightarrow{B_{\{\bullet\}}}$  is the BCurve of dimension  $D_o$ , order  $O$  and control points  $\bullet$ .  
And  $\{\vec{C}_{I(\vec{c}, d)}\}$  is the set of control points of S of indices:

$$\{I(\vec{c}, d)\} = \{ \sum_{i \in [0, D_i-1] \mid i \neq d} (O^{(D_i-1-i)} c_i) + O^{(D_i-1-d)} j \}_{j \in [0, O]} \quad (9)$$

and  $\{\vec{R}_A(\{\vec{c}\}_d, \vec{u}, d')\}$  is the set of intermediate control points calculated for:

$$\{\vec{c}\}_d = \{(\overrightarrow{c_0, c_1, \dots, c_{d-1}, j, c_{d+1}, \dots, c_{D_i-1}})\}_{j \in [0, O]} \quad (10)$$

## 2 Interface

```
// ===== BCURVE.H =====

#ifndef BCURVE_H
#define BCURVE_H

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

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"
#include "shapoid.h"

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

// ----- BCurve

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

typedef struct BCurve {
```

```

    // Order
    const int _order;
    // Dimension
    const int _dim;
    // array of (_order + 1) control points (vectors of dimension _dim)
    // defining the curve
    VecFloat** _ctrl;
} BCurve;

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

// Create a new BCurve of order 'order' and dimension 'dim'
BCurve* BCurveCreate(const int order, const int dim);

// Clone the BCurve
BCurve* BCurveClone(const BCurve* const that);

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

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

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

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

// Free the memory used by a BCurve
void BCurveFree(BCurve** that);

// Print the BCurve on 'stream'
void BCurvePrint(const BCurve* const that, FILE* const stream);

// Get the number of control point of the BCurve 'that'
#if BUILDMODE != 0
static inline
#endif
int BCurveGetNbCtrl(BCurve* const that);

// Set the value of the iCtrl-th control point to v
#if BUILDMODE != 0
static inline
#endif
void BCurveSetCtrl(BCurve* const that, const int iCtrl,
    const VecFloat* const v);

// Get a copy of the iCtrl-th control point
#if BUILDMODE != 0
static inline
#endif
VecFloat* BCurveGetCtrl(const BCurve* const that, const int iCtrl);

// Get the iCtrl-th control point
#if BUILDMODE != 0

```

```

static inline
#endif
const VecFloat* BCurveCtrl(const BCurve* const that, const int iCtrl);

// Get the 'iDim'-th value of the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
float BCurveCtrlGet(const BCurve* const that, const int iCtrl,
    const int iDim);

// Get the value of the BCurve at paramater 'u'
// u can extend beyond [0.0, 1.0]
VecFloat* BCurveGet(const BCurve* const that, const float u);

// Get the order of the BCurve
#if BUILDMODE != 0
static inline
#endif
int BCurveGetOrder(const BCurve* const that);

// Get the dimension of the BCurve
#if BUILDMODE != 0
static inline
#endif
int BCurveGetDim(const BCurve* const that);

// Get the approximate length of the BCurve (sum of dist between
// control points)
#if BUILDMODE != 0
static inline
#endif
float BCurveGetApproxLen(const BCurve* const that);

// Return the center of the BCurve (average of control points)
#if BUILDMODE != 0
static inline
#endif
VecFloat* BCurveGetCenter(const BCurve* const that);

// Rotate the curve CCW by 'theta' radians relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void BCurveRotOrigin(BCurve* const that, const float theta);

// Rotate the curve CCW by 'theta' radians relatively to its
// first control point
#if BUILDMODE != 0
static inline
#endif
void BCurveRotStart(BCurve* const that, const float theta);

// Rotate the curve CCW by 'theta' radians relatively to its
// center
#if BUILDMODE != 0
static inline
#endif
void BCurveRotCenter(BCurve* const that, const float theta);

// Scale the curve by 'v' relatively to the origin

```

```

// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleOriginVector(BCurve* const that,
    const VecFloat* const v);

// Scale the curve by 'c' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleOriginScalar(BCurve* const that, const float c);

// Scale the curve by 'v' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleStartVector(BCurve* const that, const VecFloat* const v);

// Scale the curve by 'c' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleStartScalar(BCurve* const that, const float c);

// Scale the curve by 'v' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleCenterVector(BCurve* const that,
    const VecFloat* const v);

// Scale the curve by 'c' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleCenterScalar(BCurve* const that, const float c);

// Translate the curve by 'v'
#if BUILDMODE != 0
static inline
#endif
void _BCurveTranslate(BCurve* const that, const VecFloat* const v);

// Create a BCurve which pass through the points given in the GSet 'set'
// The GSet must contains VecFloat of same dimensions
// The BCurve pass through the points in the order they are given
// in the GSet. The points don't need to be uniformly distributed
// The created BCurve is of same dimension as the VecFloat and of order
// equal to the number of VecFloat in 'set' minus one
// Return NULL if it couldn't create the BCurve
BCurve* BCurveFromCloudPoint(const GSetVecFloat* const set);

// Get a VecFloat of dimension equal to the number of control points
// Values of the VecFloat are the weight of each control point in the
// BCurve given the curve's order and the value of 't' (in [0.0,1.0])
VecFloat* BCurveGetWeightCtrlPt(const BCurve* const that, const float t);

```

```

// Get the bounding box of the BCurve.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
Facoid* BCurveGetBoundingBox(const BCurve* const that);

// ----- SCurve

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

typedef struct SCurve {
    // Order
    const int _order;
    // Dimension
    const int _dim;
    // Number of segments (one segment equals one BCurve)
    int _nbSeg;
    // Set of BCurve
    GSetBCurve _seg;
    // Set of control points
    GSetVecFloat _ctrl;
} SCurve;

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

// Create a new SCurve of dimension 'dim', order 'order' and
// 'nbSeg' segments
SCurve* SCurveCreate(const int order, const int dim, const int nbSeg);

// Clone the SCurve
SCurve* SCurveClone(const SCurve* const that);

// Return a new SCurve as a copy of the SCurve 'that' with
// dimension changed to 'dim'
// if it is extended, the values of new components are 0.0
// If it is shrunk, values are discarded from the end of the vectors
SCurve* SCurveGetNewDim(const SCurve* const that, const int dim);

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

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

// Load the SCurve from the stream
// If the SCurve is already allocated, it is freed before loading
// Return true in case of success, false else
bool SCurveLoad(SCurve** that, FILE* const stream);

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

// Free the memory used by a SCurve
void SCurveFree(SCurve** that);

// Print the SCurve on 'stream'
void SCurvePrint(const SCurve* const that, FILE* const stream);

```



```

// Get the number of BCurve in the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetNbSeg(const SCurve* const that);

// Get the dimension of the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetDim(const SCurve* const that);

// Get the order of the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetOrder(const SCurve* const that);

// Get the number of control point in the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetNbCtrl(const SCurve* const that);

// Get a clone of the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveGetCtrl(const SCurve* const that, const int iCtrl);

// Set the 'iCtrl'-th control point to a copy 'v'
#if BUILDMODE != 0
static inline
#endif
void SCurveSetCtrl(SCurve* const that, const int iCtrl,
    const VecFloat* const v);

// Set the 'iDim'-th value of the 'iCtrl'-th control point to 'v'
#if BUILDMODE != 0
static inline
#endif
void SCurveCtrlSet(SCurve* const that, const int iCtrl, const int iDim,
    float v);

// Get the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
const VecFloat* SCurveCtrl(const SCurve* const that, const int iCtrl);

// Get the 'iDim'-th value of the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
float SCurveCtrlGet(const SCurve* const that, const int iCtrl,
    const int iDim);

// Get the GSet of control points of the SCurve 'that'
#if BUILDMODE != 0
static inline
#endif
const GSetVecFloat* SCurveCtrls(const SCurve* const that);

```

```

// Get a clone of the 'iSeg'-th segment
#if BUILDMODE != 0
static inline
#endif
BCurve* SCurveGetSeg(const SCurve* const that, const int iSeg);

// Get the 'iSeg'-th segment
#if BUILDMODE != 0
static inline
#endif
const BCurve* SCurveSeg(const SCurve* const that, const int iSeg);

// Get the GSet of segments of the SCurve 'that'
#if BUILDMODE != 0
static inline
#endif
const GSetBCurve* SCurveSegs(const SCurve* const that);

// Add one segment at the end of the curve (controls are set to
// vectors null, except the first one which the last one of the current
// last segment)
void SCurveAddSegTail(SCurve* const that);

// Add one segment at the head of the curve (controls are set to
// vectors null, except the last one which the first one of the current
// first segment)
void SCurveAddSegHead(SCurve* const that);

// Remove the first segment of the curve (which must have more than one
// segment)
void SCurveRemoveHeadSeg(SCurve* const that);

// Remove the last segment of the curve (which must have more than one
// segment)
void SCurveRemoveTailSeg(SCurve* const that);

// Get the approximate length of the SCurve (sum of approxLen
// of its BCurves)
#if BUILDMODE != 0
static inline
#endif
float SCurveGetApproxLen(const SCurve* const that);

// Return the center of the SCurve (average of control points)
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveGetCenter(const SCurve* const that);

// Get the value of the SCurve at parameter 'u'
// The value is equal to the value of the floor(u)-th segment at
// value (u - floor(u))
// u can extend beyond [0.0, _nbSeg]
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveGet(const SCurve* const that, const float u);

// Return the max value for the parameter 'u' of SCurveGet
#if BUILDMODE != 0
static inline

```

```

#endif
float SCurveGetMaxU(const SCurve* const that);

// Get the bounding box of the SCurve.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
Facoid* SCurveGetBoundingBox(const SCurve* const that);

// Rotate the curve CCW by 'theta' radians relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void SCurveRotOrigin(SCurve* const that, const float theta);

// Rotate the curve CCW by 'theta' radians relatively to its
// first control point
#if BUILDMODE != 0
static inline
#endif
void SCurveRotStart(SCurve* const that, const float theta);

// Rotate the curve CCW by 'theta' radians relatively to its
// center
#if BUILDMODE != 0
static inline
#endif
void SCurveRotCenter(SCurve* const that, const float theta);

// Scale the curve by 'v' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleOriginVector(SCurve* const that,
    const VecFloat* const v);

// Scale the curve by 'c' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleOriginScalar(SCurve* const that, const float c);

// Scale the curve by 'v' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleStartVector(SCurve* const that, const VecFloat* const v);

// Scale the curve by 'c' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleStartScalar(SCurve* const that, const float c);

// Scale the curve by 'v' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline

```

```

#endif
void _SCurveScaleCenterVector(SCurve* const that,
    const VecFloat* const v);

// Scale the curve by 'c' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleCenterScalar(SCurve* const that, const float c);

// Translate the curve by 'v'
#if BUILDMODE != 0
static inline
#endif
void _SCurveTranslate(SCurve* const that, const VecFloat* const v);

// Create a new SCurve from the outline of the Shapoid 'shap'
// The Shapoid must be of dimension 2
// Control points are ordered CCW of the Shapoid
#if BUILDMODE != 0
static inline
#endif
SCurve* SCurveCreateFromShapoid(const Shapoid* const shap);

// Create a new SCurve from the outline of the Facoid 'shap'
// The Facoid must be of dimension 2
// Control points are ordered CCW of the Shapoid
SCurve* SCurveCreateFromFacoid(const Facoid* const shap);

// Create a new SCurve from the outline of the Pyramidoid 'shap'
// The Pyramidoid must be of dimension 2
// Control points are ordered CCW of the Shapoid
SCurve* SCurveCreateFromPyramidoid(const Pyramidoid* const shap);

// Create a new SCurve from the outline of the Spheroid 'shap'
// The Spheroid must be of dimension 2
// Control points are ordered CCW of the Shapoid
// Calculate an approximation as there is no exact solution
SCurve* SCurveCreateFromSpheroid(const Spheroid* const shap);

// Get the distance between the SCurve 'that' and the SCurve 'curve'
// The distance is defined as the integral of
// ||'that'(u(t))-'curve'(v(t))|| where u and v are the relative
// positions on the curve over t varying from 0.0 to 1.0
float SCurveGetDistToCurve(const SCurve* const that,
    const SCurve* const curve);

// Apply the chaikin curve subdivision algorithm to the SCurve 'that'
// with 'depth' times recursion and 'strength' is the parametric
// distance from each corner where the curve is cut at each recursion
// 'strength' in [0.0, 1.0]
// The SCurve must be of order 1, if it is not nothing happens
// cf http://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm.pdf
SCurve* SCurveChaikinSubdivision(SCurve* const that,
    const float strength, const unsigned int depth);

// ----- SCurveIter

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

typedef struct SCurveIter {

```

```

    // Attached SCurve
    const SCurve* _curve;
    // Current position
    float _curPos;
    // Step delta
    float _delta;
} SCurveIter;

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

// Create a new SCurveIter attached to the SCurve 'curve' with a step
// of 'delta'
SCurveIter SCurveIterCreateStatic(const SCurve* const curve,
    const float delta);

// Set the attached SCurve of the SCurveIter 'that' to 'curve'
#if BUILDMODE != 0
static inline
#endif
void SCurveIterSetCurve(SCurveIter* const that,
    const SCurve* const curve);

// Set the delta of the SCurveIter 'that' to 'delta'
#if BUILDMODE != 0
static inline
#endif
void SCurveIterSetDelta(SCurveIter* const that, const float delta);

// Get the attached curve of the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
const SCurve* SCurveIterCurve(const SCurveIter* const that);

// Get the delta of the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
float SCurveIterGetDelta(const SCurveIter* const that);

// Init the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
void SCurveIterInit(SCurveIter* const that);

// Step the SCurveIter 'that'
// Return false if it couldn't step, true else
#if BUILDMODE != 0
static inline
#endif
bool SCurveIterStep(SCurveIter* const that);

// Step back the SCurveIter 'that'
// Return false if it couldn't step, true else
#if BUILDMODE != 0
static inline
#endif
bool SCurveIterStepP(SCurveIter* const that);

// Get the current value of the internal parameter of the
// SCurveIter 'that'

```

```

#if BUILDMODE != 0
static inline
#endif
float SCurveIterGetPos(const SCurveIter* const that);

// Get the current value of the attached SCurve at the current
// internal position of the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveIterGet(const SCurveIter* const that);

// ----- BBody

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

typedef struct BBody {
    // Order
    const int _order;
    // Dimensions (input/output) (for example (2,3) gives a surface in 3D)
    const VecShort2D _dim;
    //  $((\_order + 1) \sim \_dim[0])$  control points of the surface
    // they are ordered as follow:
    // (0,0,0), (0,0,1), ..., (0,0,order+1), (0,1,0), (0,1,1), ...
    VecFloat** _ctrl;
} BBody;

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

// Create a new BBody of order 'order' and dimension 'dim'
// Controls are initialized with null vectors
BBody* BBodyCreate(const int order, const VecShort2D* const dim);

// Free the memory used by a BBody
void BBodyFree(BBody** that);

// Set the value of the iCtrl-th control point to v
#if BUILDMODE != 0
static inline
#endif
void _BBodySetCtrl(BBody* const that, const VecShort* const iCtrl,
    const VecFloat* const v);

// Get the value of the BBody at paramater 'u'
// u can extend beyond [0.0, 1.0]
VecFloat* _BBodyGet(const BBody* const that, const VecFloat* const u);

// Get the number of control points of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
int BBodyGetNbCtrl(const BBody* const that);

// Get the the 'iCtrl'-th control point of 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* _BBodyCtrl(const BBody* const that,
    const VecShort* const iCtrl);

// Get the index in _ctrl of the 'iCtrl' control point of 'that'
#if BUILDMODE != 0

```

```

static inline
#endif
int _BBodyGetIndexCtrl(const BBody* const that,
    const VecShort* const iCtrl);

// Get the order of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
int BBodyGetOrder(const BBody* const that);

// Get the dimensions of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* BBodyDim(const BBody* const that);

// Get a copy of the dimensions of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
VecShort2D BBodyGetDim(const BBody* const that);

// Return a clone of the BBody 'that'
BBody* BBodyClone(const BBody* const that);

// Print the BBody 'that' on the stream 'stream'
void BBodyPrint(const BBody* const that, FILE* const stream);

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

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

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

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

// Return the center of the BBody (average of control points)
#if BUILDMODE != 0
static inline
#endif
VecFloat* BBodyGetCenter(const BBody* const that);

// Translate the BBody by 'v'
#if BUILDMODE != 0
static inline
#endif
void _BBodyTranslate(BBody* const that, const VecFloat* const v);

// Scale the curve by 'v' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0

```

```

static inline
#endif
void _BBodyScaleOriginVector(BBody* const that, const VecFloat* const v);

// Scale the curve by 'c' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleOriginScalar(BBody* const that, const float c);

// Scale the curve by 'v' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleStartVector(BBody* const that, const VecFloat* const v);

// Scale the curve by 'c' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleStartScalar(BBody* const that, const float c);

// Scale the curve by 'v' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleCenterVector(BBody* const that, const VecFloat* const v);

// Scale the curve by 'c' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleCenterScalar(BBody* const that, const float c);

// Get the bounding box of the BBody.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
Facoid* BBodyGetBoundingBox(const BBody* const that);

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around 'axis'
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotAxisOrigin(BBody* const that, const VecFloat3D* const axis,
    const float theta);

// Rotate the BBody by 'theta' relatively to the center
// of the body around 'axis'
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotAxisCenter(BBody* const that, const VecFloat3D* const axis,
    const float theta);

```



```

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around 'axis'
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotAxisStart(BBody* const that, const VecFloat3D* const axis,
    const float theta);

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around X
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotXOrigin(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the center
// of the body around X
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotXCenter(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around X
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotXStart(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around Y
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotYOrigin(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the center
// of the body around Y
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotYCenter(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around Y
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotYStart(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around Z
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline

```

```

#endif
void BBodyRotZOrigin(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the center
// of the body around Z
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotZCenter(BBody* const that, const float theta);

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around Z
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotZStart(BBody* const that, const float theta);

// Create a new BBody of order 'order' which approximates best, according
// to least square regression, the point cloud defined by the
// 'inputs'/'outputs'
// 'inputs' expected in [0.0, 1.0] but may be out of range
// Return NULL if it couldn't find the BBody (the regression failed)
// If 'bias' is not null, it is set to the average of the biases from
// the least square regression on each output
// 'order' must be > 0
BBody* BBodyFromPointCloud(
    const int order,
    const GSetVecFloat* inputs,
    const GSetVecFloat* outputs,
    float* bias);

// Calculate the weights of the control points of the BBody 'that'
// for the given 'inputs'
// Return a VecFloat with weights in same order as 'that->_ctrl'
VecFloat* BBodyGetWeightCtrlPt(
    const BBody* that,
    const VecFloat* inputs);

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

#define BCurveTranslate(Curve, Vec) _Generic(Vec, \
    VecFloat*: _BCurveTranslate, \
    VecFloat2D*: _BCurveTranslate, \
    VecFloat3D*: _BCurveTranslate, \
    default: PBErrInvalidPolymorphism)(Curve, (VecFloat*)(Vec))

#define SCurveTranslate(Curve, Vec) _Generic(Vec, \
    VecFloat*: _SCurveTranslate, \
    VecFloat2D*: _SCurveTranslate, \
    VecFloat3D*: _SCurveTranslate, \
    default: PBErrInvalidPolymorphism)(Curve, (VecFloat*)(Vec))

#define BBodyTranslate(Body, Vec) _Generic(Vec, \
    VecFloat*: _BBodyTranslate, \
    VecFloat2D*: _BBodyTranslate, \
    VecFloat3D*: _BBodyTranslate, \
    default: PBErrInvalidPolymorphism)(Body, (VecFloat*)(Vec))

#define BCurveScaleOrigin(Curve, Scale) _Generic(Scale, \
    VecFloat*: _BCurveScaleOriginVector, \

```

```

float: _BCurveScaleOriginScalar, \
default: PBErrInvalidPolymorphism)(Curve, Scale)

#define BCurveScaleStart(Curve, Scale) _Generic(Scale, \
VecFloat*: _BCurveScaleStartVector, \
float: _BCurveScaleStartScalar, \
default: PBErrInvalidPolymorphism)(Curve, Scale)

#define BCurveScaleCenter(Curve, Scale) _Generic(Scale, \
VecFloat*: _BCurveScaleCenterVector, \
float: _BCurveScaleCenterScalar, \
default: PBErrInvalidPolymorphism)(Curve, Scale)

#define BBodyScaleOrigin(Body, Scale) _Generic(Scale, \
VecFloat*: _BBodyScaleOriginVector, \
float: _BBodyScaleOriginScalar, \
default: PBErrInvalidPolymorphism)(Body, Scale)

#define BBodyScaleStart(Body, Scale) _Generic(Scale, \
VecFloat*: _BBodyScaleStartVector, \
float: _BBodyScaleStartScalar, \
default: PBErrInvalidPolymorphism)(Body, Scale)

#define BBodyScaleCenter(Body, Scale) _Generic(Scale, \
VecFloat*: _BBodyScaleCenterVector, \
float: _BBodyScaleCenterScalar, \
default: PBErrInvalidPolymorphism)(Body, Scale)

#define SCurveScaleOrigin(Curve, Scale) _Generic(Scale, \
VecFloat*: _SCurveScaleOriginVector, \
float: _SCurveScaleOriginScalar, \
default: PBErrInvalidPolymorphism)(Curve, Scale)

#define SCurveScaleStart(Curve, Scale) _Generic(Scale, \
VecFloat*: _SCurveScaleStartVector, \
float: _SCurveScaleStartScalar, \
default: PBErrInvalidPolymorphism)(Curve, Scale)

#define SCurveScaleCenter(Curve, Scale) _Generic(Scale, \
VecFloat*: _SCurveScaleCenterVector, \
float: _SCurveScaleCenterScalar, \
default: PBErrInvalidPolymorphism)(Curve, Scale)

#define BBodyGetIndexCtrl(Body, ICtrl) _Generic(ICtrl, \
VecShort*: _BBodyGetIndexCtrl, \
VecShort2D*: _BBodyGetIndexCtrl, \
VecShort3D*: _BBodyGetIndexCtrl, \
VecShort4D*: _BBodyGetIndexCtrl, \
const VecShort*: _BBodyGetIndexCtrl, \
const VecShort2D*: _BBodyGetIndexCtrl, \
const VecShort3D*: _BBodyGetIndexCtrl, \
const VecShort4D*: _BBodyGetIndexCtrl, \
default: PBErrInvalidPolymorphism)(Body, (VecShort*)(ICtrl))

#define BBodyGet(Body, U) _Generic(U, \
VecFloat*: _BBodyGet, \
VecFloat2D*: _BBodyGet, \
VecFloat3D*: _BBodyGet, \
default: PBErrInvalidPolymorphism)(Body, (VecFloat*)(U))

#define BBodyCtrl(Body, ICtrl) _Generic(ICtrl, \
VecShort*: _BBodyCtrl, \

```

```

VecShort2D*: _BBodyCtrl, \
VecShort3D*: _BBodyCtrl, \
VecShort4D*: _BBodyCtrl, \
default: PBErrInvalidPolymorphism)(Body, (VecShort*)(ICtrl))

#define BBodySetCtrl(Body, ICtrl, Vec) _Generic(ICtrl, \
VecShort*: _Generic(Vec, \
VecFloat*: _BBodySetCtrl, \
VecFloat2D*: _BBodySetCtrl, \
VecFloat3D*: _BBodySetCtrl, \
default: PBErrInvalidPolymorphism), \
VecShort2D*: _Generic(Vec, \
VecFloat*: _BBodySetCtrl, \
VecFloat2D*: _BBodySetCtrl, \
VecFloat3D*: _BBodySetCtrl, \
default: PBErrInvalidPolymorphism), \
VecShort3D*: _Generic(Vec, \
VecFloat*: _BBodySetCtrl, \
VecFloat2D*: _BBodySetCtrl, \
VecFloat3D*: _BBodySetCtrl, \
default: PBErrInvalidPolymorphism), \
VecShort4D*: _Generic(Vec, \
VecFloat*: _BBodySetCtrl, \
VecFloat2D*: _BBodySetCtrl, \
VecFloat3D*: _BBodySetCtrl, \
default: PBErrInvalidPolymorphism), \
default: PBErrInvalidPolymorphism)(Body, (VecShort*)(ICtrl), \
(VecFloat*)(Vec))

// ===== static inlineer =====

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

#endif

```

## 3 Code

### 3.1 bcurve.c

```

// ===== BCURVE.C =====

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

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

// ----- BCurve

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

// Create a new BCurve of order 'order' and dimension 'dim'
BCurve* BCurveCreate(const int order, const int dim) {
#if BUILDMODE == 0

```

```

    if (order < 0) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Invalid order (%d>=0)", order);
        PBErrCatch(BCurveErr);
    }
    if (dim < 1) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Invalid dimension (%d>=1)", dim);
        PBErrCatch(BCurveErr);
    }
#endif
    // Allocate memory
    BCurve* that = PBErrMalloc(BCurveErr, sizeof(BCurve));
    // Set the values
    *((int*)&(that->_dim)) = dim;
    *((int*)&(that->_order)) = order;
    // Allocate memory for the array of control points
    that->_ctrl = PBErrMalloc(BCurveErr, sizeof(VecFloat*) * (order + 1));
    // For each control point
    for (int iCtrl = order + 1; iCtrl--;)
        // Allocate memory
        that->_ctrl[iCtrl] = VecFloatCreate(dim);
    // Return the new BCurve
    return that;
}

// Clone the BCurve
BCurve* BCurveClone(const BCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Allocate memory for the clone
    BCurve* clone = PBErrMalloc(BCurveErr, sizeof(BCurve));
    // Clone the properties
    *((int*)&(clone->_dim)) = that->_dim;
    *((int*)&(clone->_order)) = that->_order;
    // Allocate memory for the array of control points
    clone->_ctrl = PBErrMalloc(BCurveErr, sizeof(VecFloat*) *
        (clone->_order + 1));
    // For each control point
    for (int iCtrl = clone->_order + 1; iCtrl--;)
        // Clone the control point
        clone->_ctrl[iCtrl] = VecClone(that->_ctrl[iCtrl]);
    // Return the clone
    return clone;
}

// Function which return the JSON encoding of 'that'
JSONNode* BCurveEncodeAsJSON(const BCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // Create the JSON structure
    JSONNode* json = JSONCreate();

```

```

// Declare a buffer to convert value into string
char val[100];
// Encode the order
sprintf(val, "%d", BCurveGetOrder(that));
JSONAddProp(json, "_order", val);
// Encode the dimension
sprintf(val, "%d", BCurveGetDim(that));
JSONAddProp(json, "_dim", val);
// Encode the control points
JSONArrayStruct setCtrl = JSONArrayStructCreateStatic();
for (int iCtrl = 0; iCtrl < BCurveGetOrder(that) + 1; ++iCtrl)
    JSONArrayStructAdd(&setCtrl,
        VecEncodeAsJSON(BCurveCtrl(that, iCtrl)));
JSONAddProp(json, "_ctrl", &setCtrl);
// Free memory
JSONArrayStructFlush(&setCtrl);
// Return the created JSON
return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool BCurveDecodeAsJSON(BCurve** that, const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (json == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'json' is null");
        PBErrCatch(PBMathErr);
    }
#endif
    // If 'that' is already allocated
    if (*that != NULL)
        // Free memory
        BCurveFree(that);
    // Get the order from the JSON
    JSONNode* prop = JSONProperty(json, "_order");
    if (prop == NULL) {
        return false;
    }
    int order = atoi(JSONLblVal(prop));
    // Get the dimension from the JSON
    prop = JSONProperty(json, "_dim");
    if (prop == NULL) {
        return false;
    }
    int dim = atoi(JSONLblVal(prop));
    // If data are invalid
    if (order < 0 || dim < 1)
        return false;
    // Allocate memory
    *that = BCurveCreate(order, dim);
    // Decode the control points
    prop = JSONProperty(json, "_ctrl");
    if (prop == NULL) {
        return false;
    }
    if (JSONGetNbValue(prop) != order + 1) {
        return false;
    }

```

```

    }
    for (int iCtrl = 0; iCtrl < order + 1; ++iCtrl) {
        JSONNode* ctrl = JSONValue(prop, iCtrl);
        if (!VecDecodeAsJSON((*that)->_ctrl + iCtrl, ctrl) ||
            VecGetDim((*that)->_ctrl[iCtrl]) != BCurveGetDim(*that)) {
            return false;
        }
    }
    // Return the success code
    return true;
}

// Load the BCurve from the stream
// If the BCurve is already allocated, it is freed before loading
// Return true upon success, false else
bool BCurveLoad(BCurve** that, FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (stream == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'stream' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Declare a json to load the encoded data
    JSONNode* json = JSONCreate();
    // Load the whole encoded data
    if (!JSONLoad(json, stream)) {
        return false;
    }
    // Decode the data from the JSON
    if (!BCurveDecodeAsJSON(that, json)) {
        return false;
    }
    // Free the memory used by the JSON
    JSONFree(&json);
    // Return the success code
    return true;
}

// Save the BCurve to the stream
// If 'compact' equals true it saves in compact form, else it saves in
// readable form
// Return true upon success, false else
bool BCurveSave(const BCurve* const that, FILE* const stream,
    const bool compact) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (stream == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'stream' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
}

```

```

    // Get the JSON encoding
    JSONNode* json = BCurveEncodeAsJSON(that);
    // Save the JSON
    if (!JSONSave(json, stream, compact)) {
        return false;
    }
    // Free memory
    JSONFree(&jjson);
    // Return success code
    return true;
}

// Free the memory used by a BCurve
void BCurveFree(BCurve** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        return;
    // If there are control points
    if ((*that)->_ctrl != NULL)
        // For each control point
        for (int iCtrl = (*that)->_order + 1; iCtrl--;)
            // Free the control point
            VecFree((*that)->_ctrl + iCtrl);
    // Free the array of control points
    free((*that)->_ctrl);
    // Free memory
    free(*that);
    *that = NULL;
}

// Print the BCurve on 'stream'
void BCurvePrint(const BCurve* const that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (stream == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'stream' is null");
        PBErrCatch(BCurveErr);
    }
}
#endif
    // Print the order and dim
    fprintf(stream, "order(%d) dim(%d) ", that->_order, that->_dim);
    // For each control point
    for (int iCtrl = 0; iCtrl < that->_order + 1; ++iCtrl) {
        VecPrint(that->_ctrl[iCtrl], stream);
        if (iCtrl < that->_order)
            fprintf(stream, " ");
    }
}

// Get the value of the BCurve at paramater 'u'
// u can extend beyond [0.0, 1.0]
VecFloat* BCurveGet(const BCurve* const that, const float u) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }

```



```

    }
#endif
    // Allocate memory for the result
    VecFloat* v = VecFloatCreate(that->_dim);
    // Declare a variable for calcul
    VecFloat* val = VecFloatCreate(that->_order + 1);
    // Loop on dimension
    for (int dim = that->_dim; dim--;) {
        // Initialise the temporary variable with the value in current
        // dimension of the control points
        for (int iCtrl = 0; iCtrl < that->_order + 1; ++iCtrl)
            VecSet(val, iCtrl, VecGet(that->_ctrl[iCtrl], dim));
        // Loop on order
        int subOrder = that->_order;
        while (subOrder != 0) {
            // Loop on sub order
            for (int order = 0; order < subOrder; ++order)
                VecSet(val, order,
                    (1.0 - u) * VecGet(val, order) + u * VecGet(val, order + 1));
            --subOrder;
        }
        // Set the value for the current dim
        VecSet(v, dim, VecGet(val, 0));
    }
    // Free memory
    VecFree(&val);
    // Return the result
    return v;
}

// Create a BCurve which pass through the points given in the GSet 'set'
// The GSet must contains VecFloat of same dimensions
// The BCurve pass through the points in the order they are given
// in the GSet. The points don't need to be uniformly distributed
// The created BCurve is of same dimension as the VecFloat and of order
// equal to the number of VecFloat in 'set' minus one
// Return NULL if it couldn't create the BCurve
BCurve* BCurveFromCloudPoint(const GSetVecFloat* const set) {
    #if BUILDMODE == 0
        if (set == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'set' is null");
            PBErrCatch(BCurveErr);
        }
        if (GSetNbElem(set) < 1) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'set' is empty");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Declare a variable to memorize the result
    int order = GSetNbElem(set) - 1;
    long dim = VecGetDim(GSetHead(set));
    BCurve* curve = BCurveCreate(order, dim);
    // Set the first control point to the first point in the point cloud
    BCurveSetCtrl(curve, 0, GSetHead(set));
    // If the order is greater than 0
    if (order > 0) {
        // Set the last control point to the last point in the point cloud
        BCurveSetCtrl(curve, order, GSetTail(set));
        // If the order is greater than 1
        if (order > 1) {

```

```

// Calculate the t values for intermediate control points
// They are equal to the relative distance on the polyline
// linking the point in the point cloud
// Declare a variable to memorize the dimension of the matrix
// in the linear system to solve
VecShort2D dimMat = VecShortCreateStatic2D();
// Declare a variable to memorize the t values
VecFloat* t = VecFloatCreate(GSetNbElem(set));
// Set the dimensions of the matrix of the linear system
VecSet(&dimMat, 0, order - 1);
VecSet(&dimMat, 1, order - 1);
// For each point
GSetIterForward iter = GSetIterForwardCreateStatic(set);
(void)GSetIterStep(&iter);
int iPoint = 1;
do {
    // Get the distance from the previous point
    VecFloat* curPoint = GSetIterGet(&iter);
    VecFloat* prevPoint =
        GSetElemData(GSetElemPrev(GSetIterGetElem(&iter)));
    float d = VecDist(prevPoint, curPoint);
    VecSet(t, iPoint, d + VecGet(t, iPoint - 1));
    ++iPoint;
} while(GSetIterStep(&iter));
// Normalize t
for (iPoint = 1; iPoint <= order; ++iPoint)
    VecSet(t, iPoint, VecGet(t, iPoint) / VecGet(t, order));
// For each dimension
for (long iDim = dim; iDim--;) {
    // Declare a variable to memorize the matrix and vector
    // of the linear system
    MatFloat* m = MatFloatCreate(&dimMat);
    VecFloat* v = VecFloatCreate(order - 1);
    // Set the values of the linear system
    // For each line (equivalent to each intermediate point
    // in point cloud)
    for (VecSet(&dimMat, 1, 0);
        VecGet(&dimMat, 1) < order - 1;
        VecSetAdd(&dimMat, 1, 1)) {
        // Get the weight of the control point at the value
        // of t for this point
        VecFloat* weight =
            BCurveGetWeightCtrlPt(curve, VecGet(t,
                VecGet(&dimMat, 1) + 1));
        // For each intermediate control point
        for (VecSet(&dimMat, 0, 0);
            VecGet(&dimMat, 0) < order - 1;
            VecSetAdd(&dimMat, 0, 1))
            // Set the matrix value with the corresponding
            // weight
            MatSet(m, &dimMat, VecGet(weight,
                VecGet(&dimMat, 0) + 1));
        // Set the vector value with the corresponding point
        // coordinate
        float x = VecGet((VecFloat*)(GSetGet(set,
            VecGet(&dimMat, 1) + 1)), iDim);
        x -= VecGet(weight, 0) * VecGet(GSetHead(set), iDim);
        x -= VecGet(weight, order) *
            VecGet(GSetTail(set), iDim);
        VecSet(v, VecGet(&dimMat, 1), x);
    }
    // Free memory
    VecFree(&weight);
}

```

```

    }
    // Declare a variable to memorize the linear system
    SysLinEq* sys = SysLinEqCreate(m, v);
    // Solve the system
    VecFloat* solSys = SysLinEqSolve(sys);
    // If we could solve the linear system
    if (solSys != NULL) {
        // Memorize the values of control points for the
        // current dimension
        for (int iCtrl = 1; iCtrl < order; ++iCtrl)
            VecSet(curve->_ctrl[iCtrl], iDim,
                VecGet(solSys, iCtrl - 1));
        // Free memory
        VecFree(&solSys);
    } else {
        // Free memory
        SysLinEqFree(&sys);
        VecFree(&v);
        MatFree(&m);
        VecFree(&t);
        BCurveFree(&curve);
        // Return NULL
        return NULL;
    }
    // Free memory
    SysLinEqFree(&sys);
    VecFree(&v);
    MatFree(&m);
}
// Free memory
VecFree(&t);
}
}
// Return the result
return curve;
}

// Get a VecFloat of dimension equal to the number of control points
// Values of the VecFloat are the weight of each control point in the
// BCurve given the curve's order and the value of 't' (in [0.0,1.0])
VecFloat* BCurveGetWeightCtrlPt(const BCurve* const that,
    const float t) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (t < 0.0 - PBMath_EPSILON || t > 1.0 + PBMath_EPSILON) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'t' is invalid (0.0<=%f<=1.0)", t);
        PBErrCatch(BCurveErr);
    }
}
#endif
// Declare a variable to memorize the result
VecFloat* res = VecFloatCreate(that->_order + 1);
// Initilize the two first weights
VecSet(res, 0, 1.0 - t);
VecSet(res, 1, t);
// For each higher order
for (int order = 1; order < that->_order; ++order) {
    // For each control point at this order, starting by the last one

```

```

    // to avoid using a temporary buffer
    for (int iCtrl = order + 2; iCtrl-- && iCtrl != 0;)
        // Calculate the weight of this control point
        VecSet(res, iCtrl,
            (1.0 - t) * VecGet(res, iCtrl) + t * VecGet(res, iCtrl - 1));
    // Calculate the weight of the first control point
    VecSet(res, 0, (1.0 - t) * VecGet(res, 0));
}
// Return the result
return res;
}

// Get the bounding box of the BCurve.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
Facoid* BCurveGetBoundingBox(const BCurve* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Declare a variable to memorize the result
    Facoid* res = FacoidCreate(BCurveGetDim(that));
    // Create 2 vectors to memorize min and max coordinates in each
    // dimensions
    VecFloat* min = VecClone(BCurveCtrl(that, 0));
    VecFloat* max = VecClone(BCurveCtrl(that, 0));
    // For each dimension
    for (int iDim = that->_dim; iDim--;) {
        // For each control point except the first one
        for (int iCtrl = that->_order + 1; iCtrl-- && iCtrl != 0;) {
            // Update the bounding box
            if (BCurveCtrlGet(that, iCtrl, iDim) < VecGet(min, iDim))
                VecSet(min, iDim, BCurveCtrlGet(that, iCtrl, iDim));
            if (BCurveCtrlGet(that, iCtrl, iDim) > VecGet(max, iDim))
                VecSet(max, iDim, BCurveCtrlGet(that, iCtrl, iDim));
        }
    }
    // Update the result Facoid
    ShapoidSetPos(res, min);
    for (int iDim = that->_dim; iDim--;) {
        float d = VecGet(max, iDim) - VecGet(min, iDim);
        if (d < PBMath_EPSILON)
            d = 2.0 * PBMath_EPSILON;
        ShapoidAxisSet(res, iDim, iDim, d);
    }
    // Free memory
    VecFree(&min);
    VecFree(&max);
    // Return the result
    return res;
}

// ----- SCurve

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

// Create a new SCurve of dimension 'dim', order 'order' and
// 'nbSeg' segments
SCurve* SCurveCreate(const int order, const int dim, const int nbSeg) {

```

```

#if BUILDMODE == 0
    if (order < 0) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Invalid order (%d>=0)", order);
        PBErrCatch(BCurveErr);
    }
    if (dim < 1) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Invalid dimension (%d>=1)", dim);
        PBErrCatch(BCurveErr);
    }
    if (nbSeg < 1) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Invalid number of segment (%d>=1)", nbSeg);
        PBErrCatch(BCurveErr);
    }
#endif
    // Allocate memory
    SCurve* that = PBErrMalloc(BCurveErr, sizeof(SCurve));
    // Set the values
    int* d = (int*)&(that->_dim);
    *d = dim;
    int* o = (int*)&(that->_order);
    *o = order;
    that->_nbSeg = nbSeg;
    // Create the GSet
    that->_ctrl = GSetVecFloatCreateStatic();
    that->_seg = GSetBCurveCreateStatic();
    // For each segment
    for (int iSeg = nbSeg; iSeg--;) {
        // Create a segment
        BCurve* seg = BCurveCreate(order, dim);
        // If it's not the first added segment
        if (iSeg != nbSeg - 1) {
            // Replace the last control points by the current first
            VecFree(seg->_ctrl + order);
            seg->_ctrl[order] = GSetHead(&(that->_ctrl));
            // Add the control points
            for (int iCtrl = order; iCtrl--;)
                GSetPush(&(that->_ctrl), (VecFloat*)BCurveCtrl(seg, iCtrl));
            // Else, it's the first segment
        } else {
            // Add the control points
            for (int iCtrl = order + 1; iCtrl--;)
                GSetPush(&(that->_ctrl), (VecFloat*)BCurveCtrl(seg, iCtrl));
        }
        // Add the segment
        GSetPush(&(that->_seg), seg);
    }
    // Return the new SCurve
    return that;
}

// Clone the SCurve
SCurve* SCurveClone(const SCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
}
#endif

```

```

SCurve* clone = SCurveCreate(SCurveGetOrder(that), SCurveGetDim(that),
    SCurveGetNbSeg(that));
// For each control point
GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
GSetIterForward iterClone =
    GSetIterForwardCreateStatic(&(clone->_ctrl));
do {
    VecFloat* ctrl = GSetIterGet(&iter);
    VecFloat* ctrlClone = GSetIterGet(&iterClone);
    VecCopy(ctrlClone, ctrl);
} while (GSetIterStep(&iter) && GSetIterStep(&iterClone));
return clone;
}

// Return a new SCurve as a copy of the SCurve 'that' with
// dimension changed to 'dim'
// if it is extended, the values of new components are 0.0
// If it is shrinked, values are discarded from the end of the vectors
SCurve* SCurveGetNewDim(const SCurve* const that, const int dim) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (dim <= 0) {
        PBMathErr->_type = PBErrTypeInvalidArg;
        sprintf(PBMathErr->_msg, "'dim' is invalid match (%d>0)", dim);
        PBMathErrCatch(PBMathErr);
    }
#endif
    // If the new dimension is equals to the current one
    if (SCurveGetDim(that) == dim) {
        // Return the clone of the initial curve
        return SCurveClone(that);
    } else {
        // Clone the initial curve
        SCurve* ret =
            SCurveCreate(SCurveGetOrder(that), dim, SCurveGetNbSeg(that));
        // Convert the dimension of each control point
        GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
        GSetIterForward iterNew =
            GSetIterForwardCreateStatic(&(ret->_ctrl));
        do {
            VecFloat* newCtrl =
                VecGetNewDim((VecFloat*)GSetIterGet(&iter), dim);
            VecCopy((VecFloat*)(GSetIterGetElem(&iterNew)->_data), newCtrl);
            VecFree(&newCtrl);
        } while (GSetIterStep(&iter) && GSetIterStep(&iterNew));
        // Return the new curve
        return ret;
    }
}

// Function which return the JSON encoding of 'that'
JSONNode* SCurveEncodeAsJSON(const SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBMathErrCatch(PBMathErr);
    }
#endif
}

```

```

#endif
// Create the JSON structure
JSONNode* json = JSONCreate();
// Declare a buffer to convert value into string
char val[100];
// Encode the order
sprintf(val, "%d", SCurveGetOrder(that));
JSONAddProp(json, "_order", val);
// Encode the dimension
sprintf(val, "%d", SCurveGetDim(that));
JSONAddProp(json, "_dim", val);
// Encode the nb of segment
sprintf(val, "%d", SCurveGetNbSeg(that));
JSONAddProp(json, "_nbSeg", val);
// Encode the control points
JSONArrayStruct setCtrl = JSONArrayStructCreateStatic();
GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
do {
    VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
    JSONArrayStructAdd(&setCtrl, VecEncodeAsJSON(ctrl));
} while (GSetIterStep(&iter));
JSONAddProp(json, "_ctrl", &setCtrl);
// Free memory
JSONArrayStructFlush(&setCtrl);
// Return the created JSON
return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool SCurveDecodeAsJSON(SCurve** that, const JSONNode* const json) {
#if BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErrCatch(PBMathErr);
    }
    if (json == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'json' is null");
        PBErrCatch(PBMathErr);
    }
#endif
// If 'that' is already allocated
if (*that != NULL)
    // Free memory
    SCurveFree(that);
// Get the order from the JSON
JSONNode* prop = JSONProperty(json, "_order");
if (prop == NULL) {
    return false;
}
int order = atoi(JSONLblVal(prop));
// Get the dimension from the JSON
prop = JSONProperty(json, "_dim");
if (prop == NULL) {
    return false;
}
int dim = atoi(JSONLblVal(prop));
// Get the nb of segment from the JSON
prop = JSONProperty(json, "_nbSeg");
if (prop == NULL) {
    return false;
}

```

```

    }
    int nbSeg = atoi(JSONLblVal(prop));
    // If data are invalid
    if (nbSeg < 1 || order < 0 || dim < 1)
        return false;
    // Allocate memory
    *that = SCurveCreate(order, dim, nbSeg);
    // Decode the control points
    prop = JSONProperty(json, "_ctrl");
    if (prop == NULL) {
        return false;
    }
    if (JSONGetNbValue(prop) != SCurveGetNbCtrl(*that)) {
        return false;
    }
    GSetIterForward iter = GSetIterForwardCreateStatic(&((*that)->_ctrl));
    int iCtrl = 0;
    do {
        VecFloat* loadCtrl = NULL;
        JSONNode* ctrl = JSONValue(prop, iCtrl);
        if (!VecDecodeAsJSON(&loadCtrl, ctrl) ||
            VecGetDim(loadCtrl) != dim) {
            return false;
        }
        // Set the loaded control point into the set of control point
        // and segment
        VecCopy((VecFloat*)GSetIterGet(&iter), loadCtrl);
        // Free memory used by the loaded control
        VecFree(&loadCtrl);
        ++iCtrl;
    } while (GSetIterStep(&iter));
    // Return the success code
    return true;
}

// Load the SCurve from the stream
// If the SCurve is already allocated, it is freed before loading
// Return true in case of success, false else
bool SCurveLoad(SCurve** that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (stream == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'stream' is null");
        PBErrCatch(BCurveErr);
    }
}
#endif
// Declare a json to load the encoded data
JSONNode* json = JSONCreate();
// Load the whole encoded data
if (!JSONLoad(json, stream)) {
    return false;
}
// Decode the data from the JSON
if (!SCurveDecodeAsJSON(that, json)) {
    return false;
}
// Free the memory used by the JSON

```



```

    JSONFree(&json);
    // Return the success code
    return true;
}

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

// Free the memory used by a SCurve
void SCurveFree(SCurve** that) {
    // Check argument
    if (that == NULL || *that == NULL)
        return;
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&((*that)->_ctrl));
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        // Free the memory used by the control point
        VecFree(&ctrl);
    } while (GSetIterStep(&iter));
    // Free the memory used by the set of control point
    GSetFlush(&((*that)->_ctrl));
    // For each segment
    iter = GSetIterForwardCreateStatic(&((*that)->_seg));
    do {
        BCurve* seg = (BCurve*)GSetIterGet(&iter);
        // Disconnect the control points which have been already freed
        // or doesn't need to be freed (the last one)
        for (int iCtrl = 0; iCtrl <= (*that)->_order; ++iCtrl)
            seg->_ctrl[iCtrl] = NULL;
        // Free the meory used by the segment
        BCurveFree(&seg);
    } while (GSetIterStep(&iter));
    // Free the memory used by the set of segment
    GSetFlush(&((*that)->_seg));
}

```

```

    // Free memory
    free(*that);
    *that = NULL;
}

// Print the SCurve on 'stream'
void SCurvePrint(const SCurve* const that, FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (stream == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'stream' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Print the order and dim
    fprintf(stream, "order(%d) dim(%d) nbSeg(%d) ",
        that->_order, that->_dim, that->_nbSeg);
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    int iMark = 0;
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        if (iMark == 0)
            fprintf(stream, "<");
        //VecPrint(ctrl, stream);
        VecFloatPrint(ctrl, stream, 6);
        if (iMark == 0)
            fprintf(stream, ">");
        if (GSetIterIsLast(&iter) == false)
            fprintf(stream, " ");
        ++iMark;
        if (iMark == that->_order)
            iMark = 0;
    } while (GSetIterStep(&iter));
}

// Add one segment at the end of the curve (controls are set to
// vectors null, except the first one which the last one of the current
// last segment)
void SCurveAddSegTail(SCurve* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Create the new segment
    BCurve* seg = BCurveCreate(that->_order, that->_dim);
    // Free memory used by the first control point
    VecFree(seg->_ctrl);
    // Replace it with the current last control
    seg->_ctrl[0] = GSetTail(&(that->_ctrl));
    // Add the segment to the set of segment
    GSetAppend(&(that->_seg), seg);
    // Add the new control points to the set of control points
    for (int iCtrl = 1; iCtrl <= that->_order; ++iCtrl)

```

```

        GSetAppend(&(that->_ctrl), seg->_ctrl[iCtrl]);
    // Update the number of segment
    ++(that->_nbSeg);
}

// Add one segment at the head of the curve (controls are set to
// vectors null, except the last one which the first one of the current
// first segment)
void SCurveAddSegHead(SCurve* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Create the new segment
    BCurve* seg = BCurveCreate(that->_order, that->_dim);
    // Free memory used by the last control point
    VecFree(seg->_ctrl + that->_order);
    // Replace it with the current first control
    seg->_ctrl[that->_order] = GSetHead(&(that->_ctrl));
    // Add the segment to the set of segment
    GSetPush(&(that->_seg), seg);
    // Add the new control points to the set of control points
    for (int iCtrl = that->_order; iCtrl--;)
        GSetPush(&(that->_ctrl), seg->_ctrl[iCtrl]);
    // Update the number of segment
    ++(that->_nbSeg);
}

// Remove the first segment of the curve (which must have more than one
// segment)
void SCurveRemoveHeadSeg(SCurve* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (that->_nbSeg < 2) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'that' has only one segment");
            PBErrCatch(BCurveErr);
        }
    #endif
    // Remove the control points from the set of control points
    for (int iCtrl = 0; iCtrl < that->_order; ++iCtrl) {
        VecFloat* ctrl = (VecFloat*)GSetPop(&(that->_ctrl));
        VecFree(&ctrl);
    }
    // Remove the first segment
    BCurve* seg = (BCurve*)GSetPop(&(that->_seg));
    // Disconnect the control points which have been already freed
    // or doesn't need to be freed (the last one)
    for (int iCtrl = 0; iCtrl <= that->_order; ++iCtrl)
        seg->_ctrl[iCtrl] = NULL;
    // Free the memory used by the segment
    BCurveFree(&seg);
    // Update the number of segment
    --(that->_nbSeg);
}

```

```

// Remove the last segment of the curve (which must have more than one
// segment)
void SCurveRemoveTailSeg(SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (that->_nbSeg < 2) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'that' has only one segment");
        PBErrCatch(BCurveErr);
    }
#endif
    // Remove the control points from the set of control points
    for (int iCtrl = 0; iCtrl < that->_order; ++iCtrl) {
        VecFloat* ctrl = (VecFloat*)GSetDrop(&(that->_ctrl));
        VecFree(&ctrl);
    }
    // Remove the last segment
    BCurve* seg = (BCurve*)GSetDrop(&(that->_seg));
    // Disconnect the control points which have been already freed
    // or doesn't need to be freed (the first one)
    for (int iCtrl = 0; iCtrl <= that->_order; ++iCtrl)
        seg->_ctrl[iCtrl] = NULL;
    // Free the memory used by the segment
    BCurveFree(&seg);
    // Update the number of segment
    --(that->_nbSeg);
}

// Get the bounding box of the SCurve.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
// TODO : better solution possible, refer to
// https://pomax.github.io/bezierinfo/#circles_cubic
Facoid* SCurveGetBoundingBox(const SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Declare a set to memorize the bounding box of each segment
    GSetShapoid set = GSetShapoidCreateStatic();
    // For each segment
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_seg));
    do {
        // Add the bounding box of this segment to the set
        GSetPush(&set,
            BCurveGetBoundingBox((BCurve*)GSetIterGet(&iter)));
    } while (GSetIterStep(&iter));
    // Get the bounding box of all the segment's bounding box
    Facoid* bound = ShapoidGetBoundingBoxSet(&set);
    // Free the memory used by the bounding box of each segment
    iter = GSetIterForwardCreateStatic(&set);
    do {
        Facoid* facoid = (Facoid*)GSetIterGet(&iter);
        ShapoidFree(&facoid);
    }

```

```

    } while (GSetIterStep(&iter));
    GSetFlush(&set);
    // Return the bounding box
    return bound;
}

// Create a new SCurve from the outline of the Facoid 'shap'
// The Facoid must be of dimension 2
// Control points are ordered CCW of the Shapoid
SCurve* SCurveCreateFromFacoid(const Facoid* const shap) {
#ifdef BUILDMODE == 0
    if (shap == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'shap' is null");
        PBErrCatch(BCurveErr);
    }
    if (ShapoidGetDim(shap) != 2) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg,
            "'shap' 's dimension is invalid (%d==2)",
            ShapoidGetDim(shap));
        PBErrCatch(BCurveErr);
    }
#endif
    // Create the curve
    int order = 1;
    int dim = 2;
    int nbSeg = 4;
    SCurve* ret = SCurveCreate(order, dim, nbSeg);
    // Set the coordinates of the control points according to the
    // Facoid
    VecFloat* v = VecClone(ShapoidPos(shap));
    SCurveSetCtrl(ret, 0, v);
    SCurveSetCtrl(ret, 4, v);
    VecOp(v, 1.0, ShapoidAxis(shap, 0), 1.0);
    SCurveSetCtrl(ret, 1, v);
    VecOp(v, 1.0, ShapoidAxis(shap, 1), 1.0);
    SCurveSetCtrl(ret, 2, v);
    VecOp(v, 1.0, ShapoidAxis(shap, 0), -1.0);
    SCurveSetCtrl(ret, 3, v);
    VecFree(&v);
    // Return the curve
    return ret;
}

// Create a new SCurve from the outline of the Pyramidoid 'shap'
// The Pyramidoid must be of dimension 2
// Control points are ordered CCW of the Shapoid
SCurve* SCurveCreateFromPyramidoid(const Pyramidoid* const shap) {
#ifdef BUILDMODE == 0
    if (shap == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'shap' is null");
        PBErrCatch(BCurveErr);
    }
    if (ShapoidGetDim(shap) != 2) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg,
            "'shap' 's dimension is invalid (%d==2)",
            ShapoidGetDim(shap));
        PBErrCatch(BCurveErr);
    }
#endif
}

```

```

#endif
// Create the curve
SCurve* ret = SCurveCreate(1, 2, 3);
// Set the coordinates of the control points according to the
// Facoid
VecFloat* ctrl[2] = {NULL};
for (int i = 2; i--;) {
    ctrl[i] = VecClone(ShapoidPos(shap));
}
VecOp(ctrl[0], 1.0, ShapoidAxis(shap, 0), 1.0);
VecOp(ctrl[1], 1.0, ShapoidAxis(shap, 1), 1.0);
SCurveSetCtrl(ret, 0, ShapoidPos(shap));
SCurveSetCtrl(ret, 1, ctrl[0]);
SCurveSetCtrl(ret, 2, ctrl[1]);
SCurveSetCtrl(ret, 3, ShapoidPos(shap));
for (int i = 2; i--;)
    VecFree(ctrl + i);
// Return the curve
return ret;
}

// Create a new SCurve from the outline of the Spheroid 'shap'
// The Spheroid must be of dimension 2
// Control points are ordered CCW of the Shapoid
// Calculate an approximation as there is no exact solution
SCurve* SCurveCreateFromSpheroid(const Spheroid* const shap) {
    #if BUILDMODE == 0
        if (shap == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'shap' is null");
            PBErrCatch(BCurveErr);
        }
        if (ShapoidGetDim(shap) != 2) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg,
                "'shap' 's dimension is invalid (%d==2)",
                ShapoidGetDim(shap));
            PBErrCatch(BCurveErr);
        }
    #endif
    // Create the curve
    SCurve* ret = SCurveCreate(3, 2, 4);
    // Set the control points
    // The anchors of the curve can be easily calculated from the
    // position and axis of the Spheroid
    int iAxis = 0;
    float coeff = 0.5;
    VecFloat* ctrl[13] = {NULL};
    for (int i = 13; i--;)
        ctrl[i] = VecFloatCreate(2);
    for (int i = 0; i < 12; i += 3) {
        VecCopy(ctrl[i], ShapoidPos(shap));
        if (i == 6)
            coeff *= -1.0;
        VecOp(ctrl[i], 1.0, ShapoidAxis(shap, iAxis), coeff);
        if (i > 0)
            VecCopy(ctrl[i - 1], ctrl[i]);
        if (i < 11)
            VecCopy(ctrl[i + 1], ctrl[i]);
        iAxis = (iAxis == 0 ? 1 : 0);
    }
    VecCopy(ctrl[12], ctrl[0]);

```

```

VecCopy(ctrl[11], ctrl[0]);
// Calculate the others control points by transforming the
// quadratic approximation of a quarter of the unit circle :
// A(1,0), B(1,4(sqrt(2)-1)/3), C(4(sqrt(2)-1)/3,1), D(0,1)
// toward the Spheroid
float c = 0.276142;
VecOp(ctrl[1], 1.0, ShapoidAxis(shap, 1), c);
VecOp(ctrl[2], 1.0, ShapoidAxis(shap, 0), c);
VecOp(ctrl[4], 1.0, ShapoidAxis(shap, 0), -1.0 * c);
VecOp(ctrl[5], 1.0, ShapoidAxis(shap, 1), c);
VecOp(ctrl[7], 1.0, ShapoidAxis(shap, 1), -1.0 * c);
VecOp(ctrl[8], 1.0, ShapoidAxis(shap, 0), -1.0 * c);
VecOp(ctrl[10], 1.0, ShapoidAxis(shap, 0), c);
VecOp(ctrl[11], 1.0, ShapoidAxis(shap, 1), -1.0 * c);
for (int i = 13; i--;)
    SCurveSetCtrl(ret, i, ctrl[i]);
for (int i = 13; i--;)
    VecFree(ctrl + i);
// Return the curve
return ret;
}

// Get the distance between the SCurve 'that' and the SCurve 'curve'
// The distance is defined as the integral of
// ||'that'(u(t))-'curve'(v(t))|| where u and v are the relative
// positions on the curve over t varying from 0.0 to 1.0
float SCurveGetDistToCurve(const SCurve* const that,
    const SCurve* const curve) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'shap' is null");
        PBErrCatch(BCurveErr);
    }
    if (curve == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'shap' is null");
        PBErrCatch(BCurveErr);
    }
    if (SCurveGetDim(that) != SCurveGetDim(curve)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg,
            "dimensions of 'that' and 'curve' differ (%d==%d)",
            SCurveGetDim(that), SCurveGetDim(curve));
        PBErrCatch(BCurveErr);
    }
#endif
    // Declare a variable to memorize the result
    float res = 0.0;
    // Declare a variable to memorize the step over parameter
    float dt = 0.01;
    int nb = (int)floor(1.0 / dt);
    float t = 0.0;
    // Loop over the parameter
    for (int i = nb; i--;) {
        // Calculate the relative parameter for both curves
        float u = t * SCurveGetMaxU(that);
        float v = t * SCurveGetMaxU(curve);
        // Get the value of both curve at these relative parameters
        VecFloat* valA = SCurveGet(that, u);
        VecFloat* valB = SCurveGet(curve, v);
        // Get the distance between value

```

```

    float dist = VecDist(valA, valB);
    // Add to result
    res += dist * dt;
    // Step the parameter
    t += dt;
    // Free memory
    VecFree(&valA);
    VecFree(&valB);
}
// Return the result
return res;
}

// Apply the chaikin curve subdivision algorithm to the SCurve 'that'
// with 'depth' times recursion and 'strength' is the parametric
// distance from each corner where the curve is cut at each recursion
// 'strength' in [0.0, 1.0]
// The SCurve must be of order 1, if it is not nothing happens
// cf http://graphics.cs.ucdavis.edu/education/CAGDNotes/Chaikins-Algorithm.pdf
SCurve* SCurveChaikinSubdivision(SCurve* const that,
    const float strength, const unsigned int depth) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (strength < 0.0 || strength > 1.0) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'strength' is invalid (0<=%f<=1)",
            strength);
        PBErrCatch(BCurveErr);
    }
}
#endif

// Init the result with a clone of the original curve
SCurve* res = SCurveClone(that);

// If the SCurve order is 0
if (SCurveGetOrder(that) == 1) {

    // Loop on the depth
    for (unsigned int iDepth = depth; iDepth--;) {

        // Declare the new set of BCurve
        GSetBCurve segs = GSetBCurveCreateStatic();

        // Declare the new set of control points
        GSetVecFloat ctrls = GSetVecFloatCreateStatic();

        // Declare a variable to memorize the new segment
        BCurve* newSeg = NULL;

        // Loop on the segments of the curve
        GSetIterForward iter = GSetIterForwardCreateStatic(
            SCurveSegs(res));
        do {

            // Get the current segment
            BCurve* curSeg = GSetIterGet(&iter);

            // If it's the first segment

```



```

if (GSetIterIsFirst(&iter)) {

    // Get the cut position
    VecFloat* pos = BCurveGet(curSeg, 1.0 - strength);

    // Create the new segment for the head of the current segment
    newSeg = BCurveCreate(SCurveGetOrder(that),
        SCurveGetDim(that));
    VecFree(newSeg->_ctrl);
    VecFree(newSeg->_ctrl + 1);
    newSeg->_ctrl[0] = VecClone(curSeg->_ctrl[0]);
    newSeg->_ctrl[1] = pos;

    // Add the new segment
    GSetAppend(&segs, newSeg);
    GSetAppend(&ctrls, newSeg->_ctrl[0]);
    GSetAppend(&ctrls, newSeg->_ctrl[1]);

    // Create the new segment for the tail of the current segment
    // The second control of this segment will be set later
    newSeg = BCurveCreate(SCurveGetOrder(that),
        SCurveGetDim(that));
    VecFree(newSeg->_ctrl);
    VecFree(newSeg->_ctrl + 1);
    newSeg->_ctrl[0] = pos;

    // Add the new segment
    GSetAppend(&segs, newSeg);

} else if (GSetIterIsLast(&iter)) {

    // Get the cut position
    VecFloat* pos = BCurveGet(curSeg, strength);

    // Set the control of the last new segment
    newSeg->_ctrl[1] = pos;
    GSetAppend(&ctrls, newSeg->_ctrl[1]);

    // Create the new segment for the tail of the current segment
    // The second control of this segment will be set later
    newSeg = BCurveCreate(SCurveGetOrder(that),
        SCurveGetDim(that));
    VecFree(newSeg->_ctrl);
    VecFree(newSeg->_ctrl + 1);
    newSeg->_ctrl[0] = pos;
    newSeg->_ctrl[1] = VecClone(curSeg->_ctrl[1]);

    // Add the new segment
    GSetAppend(&segs, newSeg);
    GSetAppend(&ctrls, newSeg->_ctrl[1]);

} else {

    // Else, it's a segment inside the curve

    // Get the cut positions
    VecFloat* posA = BCurveGet(curSeg, strength);
    VecFloat* posB = BCurveGet(curSeg, 1.0 - strength);

    // Set the second control of the last new segment
    newSeg->_ctrl[1] = posA;
    GSetAppend(&ctrls, newSeg->_ctrl[1]);
}

```

```

        // Create the new segment for the center of the current segment
        newSeg = BCurveCreate(SCurveGetOrder(that),
            SCurveGetDim(that));
        VecFree(newSeg->_ctrl);
        VecFree(newSeg->_ctrl + 1);
        newSeg->_ctrl[0] = posA;
        newSeg->_ctrl[1] = posB;

        // Add the new segment
        GSetAppend(&segs, newSeg);
        GSetAppend(&ctrls, newSeg->_ctrl[1]);

        // Create the new segment for the tail of the current segment
        // The second control of this segment will be set later
        newSeg = BCurveCreate(SCurveGetOrder(that),
            SCurveGetDim(that));
        VecFree(newSeg->_ctrl);
        VecFree(newSeg->_ctrl + 1);
        newSeg->_ctrl[0] = posB;

        // Add the new segment
        GSetAppend(&segs, newSeg);

    }
} while (GSetIterStep(&iter));

// Free the current result
SCurveFree(&res);

// Create the new result SCurve
res = SCurveCreate(
    SCurveGetOrder(that), SCurveGetDim(that), 1);
GSetFlush(&(res->_ctrl));
BCurve* curve = GSetPop((GSet*)SCurveSegs(res));
BCurveFree(&curve);
res->_seg = segs;
res->_ctrl = ctrls;
res->_nbSeg = GSetNbElem(SCurveSegs(res));

}
}

// Return the result
return res;
}

// ----- SCurveIter

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

// Create a new SCurveIter attached to the SCurve 'curve' with a step
// of 'delta'
SCurveIter SCurveIterCreateStatic(const SCurve* const curve,
    const float delta) {
    #if BUILDMODE == 0
        if (curve == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'curve' is null");
            PBErrCatch(BCurveErr);
        }
        if (delta <= 0.0) {

```

```

        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'delta' is invalid (%f>0)", delta);
        PBErrCatch(BCurveErr);
    }
#endif
    // Declare the new SCurveIter
    SCurveIter iter;
    // Set the properties
    iter._curve = curve;
    iter._curPos = 0.0;
    iter._delta = delta;
    // Return the new iterator
    return iter;
}

// ----- BBody

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

// Create a new BBody of order 'order' and dimension 'dim'
// Controls are initialized with null vectors
BBody* BBodyCreate(const int order, const VecShort2D* const dim) {
    #if BUILDMODE == 0
        if (order < 0) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "Invalid order (%d>=0)", order);
            PBErrCatch(BCurveErr);
        }
        if (dim == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'dim' is null");
            PBErrCatch(BCurveErr);
        }
        for (int iDim = 2; iDim--;) {
            if (VecGet(dim, iDim) <= 0) {
                BCurveErr->_type = PBErrTypeInvalidArg;
                sprintf(BCurveErr->_msg, "Dimension is invalid (dim[%d]:%d>0)",
                    iDim, VecGet(dim, iDim));
                PBErrCatch(BCurveErr);
            }
        }
    }
#endif
    // Allocate memory for the new BBody
    BBody* that = PBErrMalloc(BCurveErr, sizeof(BBody));
    // Init pointers
    *((VecShort2D*)&(that->_dim)) = VecShortCreateStatic2D();
    that->_ctrl = NULL;
    // Init properties
    *((int*)&(that->_order)) = order;
    *((VecShort2D*)&(that->_dim)) = *dim;
    // Init the control
    int nbCtrl = BBodyGetNbCtrl(that);
    that->_ctrl = PBErrMalloc(BCurveErr, sizeof(VecFloat*) * nbCtrl);
    for (int iCtrl = nbCtrl; iCtrl--;)
        that->_ctrl[iCtrl] = VecFloatCreate(VecGet(dim, 1));
    // Return the new BBody
    return that;
}

// Free the memory used by a BBody
void BBodyFree(BBody** that) {
    // Check arguments

```

```

    if (that == NULL || *that == NULL)
        return;
    // Get the number of ctrl
    int nbCtrl = BBodyGetNbCtrl(*that);
    // Free memory
    for (int iCtrl = nbCtrl; iCtrl--;)
        VecFree((*that)->_ctrl + iCtrl);
    free((*that)->_ctrl);
    free(*that);
    *that = NULL;
}

// Get the value of the BBody at paramater 'u'
// u can extend beyond [0.0, 1.0]
VecFloat* _BBodyGet(const BBody* const that, const VecFloat* const u) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (u == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'u' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(u) != VecGet(&(that->_dim), 0)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'u' is invalid (%ld=%d)",
            VecGetDim(u), VecGet(&(that->_dim), 0));
        PBErrCatch(BCurveErr);
    }
#endif
    // Get the weights of the control
    VecFloat* weights =
        BBodyGetWeightCtrlPt(
            that,
            u);

    // Allocate memory for the result
    int nbDimOut =
        VecGet(
            &(that->_dim),
            1);
    VecFloat* res = VecFloatCreate(nbDimOut);

    // Get the result. This is the sum of the control points multiplied
    // by their coefficient
    int nbCtrl = BBodyGetNbCtrl(that);
    for (
        int iCtrl = nbCtrl;
        iCtrl--;) {

        float weight =
            VecGet(
                weights,
                iCtrl);
        VecOp(
            res,
            1.0,
            that->_ctrl[iCtrl],
            weight);
    }
}

```

```

    }

    // Free memory
    VecFree(&weights);

    // Return the result
    return res;
}

// Return a clone of the BBody 'that'
BBody* BBodyClone(const BBody* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Declare the clone
    BBody* clone = BBodyCreate(BBodyGetOrder(that), BBodyDim(that));
    // For each control
    for (int iCtrl = BBodyGetNbCtrl(clone); iCtrl--;)
        // Copy the control values
        VecCopy(clone->_ctrl[iCtrl], that->_ctrl[iCtrl]);
    // Return the clone
    return clone;
}

// Print the BBody 'that' on the stream 'stream'
void BBodyPrint(const BBody* const that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (stream == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'stream' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Print the order and dim
    fprintf(stream, "order(%d) dim(", that->_order);
    VecPrint(&(that->_dim), stream);
    fprintf(stream, ") ");
    // For each control point
    for (int iCtrl = 0; iCtrl < BBodyGetNbCtrl(that); ++iCtrl) {
        VecPrint(that->_ctrl[iCtrl], stream);
        if (iCtrl < that->_order)
            fprintf(stream, " ");
    }
}

// Function which return the JSON encoding of 'that'
JSONNode* BBodyEncodeAsJSON(const BBody* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErrTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
    }
#endif
}

```

```

        PBErCatch(PBMathErr);
    }
#endif
    // Create the JSON structure
    JSONNode* json = JSONCreate();
    // Declare a buffer to convert value into string
    char val[100];
    // Encode the order
    sprintf(val, "%d", BBodyGetOrder(that));
    JSONAddProp(json, "_order", val);
    // Encode the dimension
    JSONAddProp(json, "_dim", VecEncodeAsJSON((VecShort*)BBodyDim(that)));
    // Encode the control points
    JSONArrayStruct setCtrl = JSONArrayStructCreateStatic();
    // For each control point
    for (int iCtrl = 0; iCtrl < BBodyGetNbCtrl(that); ++iCtrl)
        JSONArrayStructAdd(&setCtrl, VecEncodeAsJSON(that->_ctrl[iCtrl]));
    JSONAddProp(json, "_ctrl", &setCtrl);
    // Free memory
    JSONArrayStructFlush(&setCtrl);
    // Return the created JSON
    return json;
}

// Function which decode from JSON encoding 'json' to 'that'
bool BBodyDecodeAsJSON(BBody** that, const JSONNode* const json) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'that' is null");
        PBErCatch(PBMathErr);
    }
    if (json == NULL) {
        PBMathErr->_type = PBErTypeNullPointer;
        sprintf(PBMathErr->_msg, "'json' is null");
        PBErCatch(PBMathErr);
    }
#endif
    // If 'that' is already allocated
    if (*that != NULL)
        // Free memory
        BBodyFree(that);
    // Get the order from the JSON
    JSONNode* prop = JSONProperty(json, "_order");
    if (prop == NULL) {
        return false;
    }
    int order = atoi(JSONLblVal(prop));
    // Get the dimension from the JSON
    prop = JSONProperty(json, "_dim");
    if (prop == NULL) {
        return false;
    }
    VecShort* dim = NULL;
    if (!VecDecodeAsJSON(&dim, prop)) {
        return false;
    }
    // If data are invalid
    if (order < 0 || VecGetDim(dim) != 2 ||
        VecGet(dim, 0) < 1 || VecGet(dim, 1) < 1) {
        return false;
    }
}

```

```

// Allocate memory
*that = BBodyCreate(order, (VecShort2D*)dim);
// Decode the control points
prop = JSONProperty(json, "_ctrl");
if (prop == NULL) {
    return false;
}
if (JSONGetNbValue(prop) != BBodyGetNbCtrl(*that)) {
    return false;
}
for (int iCtrl = 0; iCtrl < BBodyGetNbCtrl(*that); ++iCtrl) {
    JSONNode* ctrl = JSONValue(prop, iCtrl);
    if (!VecDecodeAsJSON((*that)->_ctrl + iCtrl, ctrl))
        return false;
    // If the control point is not of the correct dimension
    if (VecGetDim((*that)->_ctrl[iCtrl]) != VecGet(&((*that)->_dim), 1))
        return false;
}
// Free memory
VecFree(&dim);
// Return the success code
return true;
}

// Load the BBody from the stream
// If the BBody is already allocated, it is freed before loading
// Return true upon success, false else
bool BBodyLoad(BBody** that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (stream == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'stream' is null");
        PBErrCatch(BCurveErr);
    }
}
#endif
// Declare a json to load the encoded data
JSONNode* json = JSONCreate();
// Load the whole encoded data
if (!JSONLoad(json, stream)) {
    return false;
}
// Decode the data from the JSON
if (!BBodyDecodeAsJSON(that, json)) {
    return false;
}
// Free the memory used by the JSON
JSONFree(&json);
// Return the success code
return true;
}

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

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (stream == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'stream' is null");
        PBErrCatch(BCurveErr);
    }
#endif
// Get the JSON encoding
JSONNode* json = BBodyEncodeAsJSON(that);
// Save the JSON
if (!JSONSave(json, stream, compact)) {
    return false;
}
// Free memory
JSONFree(&jjson);
// Return success code
return true;
}

// Get the bounding box of the BBody.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
Facoid* BBodyGetBoundingBox(const BBody* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
// Declare a variable to memorize the result
Facoid* res = FacoidCreate(VecGet(BBodyDim(that), 1));
// For each dimension
for (int iDim = VecGet(BBodyDim(that), 1); iDim--;) {
    // Initialise the bounding box in this dimension
    VecSet(res->s._pos, iDim, VecGet(that->_ctrl[0], iDim));
    VecSet(res->s._axis[iDim], iDim, VecGet(that->_ctrl[0], iDim));
    // For each control point except the first one
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
        // Update the bounding box
        if (VecGet(that->_ctrl[iCtrl], iDim) < VecGet(res->s._pos, iDim))
            VecSet(res->s._pos, iDim, VecGet(that->_ctrl[iCtrl], iDim));
        if (VecGet(that->_ctrl[iCtrl], iDim) >
            ShapoidAxisGet(res, iDim, iDim))
            ShapoidAxisSet(res, iDim, iDim,
                VecGet(that->_ctrl[iCtrl], iDim));
    }
    ShapoidAxisSetAdd(res, iDim, iDim, -1.0 * ShapoidPosGet(res, iDim));
}
// Return the result
return res;
}

// Create a new BBody of order 'order' which approximates best, according
// to least square regression, the point cloud defined by the
// 'inputs'/'outputs'
// 'inputs' expected in [0.0, 1.0] but may be out of range

```



```

// Return NULL if it couldn't find the BBody (the regression failed)
// If 'bias' is not null, it is set to the average of the biases from
// the least square regression on each output
// 'order' must be > 0
BBody* BBodyFromPointCloud(
    const int order,
    const GSetVecFloat* inputs,
    const GSetVecFloat* outputs,
    float* bias) {

#ifdef BUILDMODE == 0

    if (inputs == NULL) {

        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(
            BCurveErr->_msg,
            "'inputs' is null");
        PBErrCatch(BCurveErr);

    }

    if (outputs == NULL) {

        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(
            BCurveErr->_msg,
            "'outputs' is null");
        PBErrCatch(BCurveErr);

    }

    if (order < 1) {

        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(
            BCurveErr->_msg,
            "Invalid order (%d>=1)",
            order);
        PBErrCatch(BCurveErr);

    }

    if (GSetNbElem((GSet*)inputs) == 0) {

        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(
            BCurveErr->_msg,
            "Empty inputs");
        PBErrCatch(BCurveErr);

    }

    if (GSetNbElem((GSet*)outputs) == 0) {

        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(
            BCurveErr->_msg,
            "Empty outputs");
        PBErrCatch(BCurveErr);

    }

}

```

```

if (GSetNbElem((GSet*)inputs) != GSetNbElem((GSet*)outputs)) {

    BCurveErr->_type = PErrTypeInvalidArg;
    sprintf(
        BCurveErr->_msg,
        "Different number of inputs and outputs (%ld==%ld)",
        GSetNbElem((GSet*)inputs),
        GSetNbElem((GSet*)outputs));
    PErrCatch(BCurveErr);

}

#endif

// Get the number and dimensions of inputs and outputs
long dimInputs = VecGetDim((VecFloat*)GSetHead((GSet*)inputs));
long dimOutputs = VecGetDim((VecFloat*)GSetHead((GSet*)outputs));
unsigned int nbPoints = GSetNbElem((GSet*)inputs);

// Create the BBody
VecShort2D dim = VecShortCreateStatic2D();
VecSet(
    &dim,
    0,
    dimInputs);
VecSet(
    &dim,
    1,
    dimOutputs);
BBody* res =
    BBodyCreate(
        order,
        &dim);

// Allocate memory for the left hand side of the system for the
// regression
VecShort2D v = VecShortCreateStatic2D();
VecSet(
    &v,
    0,
    BBodyGetNbCtrl(res));
VecSet(
    &v,
    1,
    nbPoints);
MatFloat* X = MatFloatCreate(&v);

// Loop on the points in the point cloud
GSetIterForward iter = GSetIterForwardCreateStatic((GSet*)inputs);
bool flagStep = true;
unsigned int iPoint = 0;
do {

    // Get the input
    VecFloat* input = GSetIterGet(&iter);

    // Set the index of the row in the matrix for this point
    VecSet(
        &v,
        1,
        iPoint);

```

```

// Get the weights of the control points for the inputs of
// this point
VecFloat* weights =
    BBodyGetWeightCtrlPt(
        res,
        input);

// Loop on the control points (i.e. the column of the regression
// matrix)
for (
    int iCtrl = BBodyGetNbCtrl(res);
    iCtrl--;) {

    // Get the weight of this control point for this point in the
    // point cloud
    float weight =
        VecGet(
            weights,
            iCtrl);

    // Set the index of the column in the matrix
    VecSet(
        &v,
        0,
        iCtrl);

    // Set the weight in the matrix
    MatSet(
        X,
        &v,
        weight);

}

// Free memory
VecFree(&weights);

// Step to the next point
++iPoint;
flagStep = GSetIterStep(&iter);

} while (flagStep);

// Create the LeastSquareLinReg
LeastSquareLinReg lslr = LeastSquareLinRegCreateStatic(X);

// If the regression has no solution
if (LSLRIsSolvable(&lslr) == false) {

    // Stop here
    LeastSquareLinRegFreeStatic(&lslr);
    MatFree(&X);
    BBodyFree(&res);
    return NULL;

}

// If the user requested the bias
if (bias != NULL) {

    // Initialise the bias

```

```

    *bias = 0.0;
}

// Loop on the outputs dimension
for (
    long iOut = 0;
    iOut < dimOutputs;
    ++iOut) {

    // Create the right hand side of the system for the regression
    // by extracting the iOut-th component of the outputs
    VecFloat* Y = VecFloatCreate(nbPoints);
    GSetIterForward iter = GSetIterForwardCreateStatic((GSet*)outputs);
    flagStep = true;
    iPoint = 0;
    do {

        // Get the output
        VecFloat* output = GSetIterGet(&iter);

        float val =
            VecGet(
                output,
                iOut);
        VecSet(
            Y,
            iPoint,
            val);

        // Step to the next point
        ++iPoint;
        flagStep = GSetIterStep(&iter);

    } while (flagStep);

    // Solve the regression
    VecFloat* beta =
        LSLRSolve(
            &lslr,
            Y);

    // Loop on the control points
    for (
        int iCtrl = BBodyGetNbCtrl(res);
        iCtrl--;) {

        // Update the iOut-th component of the control points with the
        // result of the regression
        float val =
            VecGet(
                beta,
                iCtrl);
        VecSet(
            res->_ctrl[iCtrl],
            iOut,
            val);

    }

    // If the user requested the bias
    if (bias != NULL) {

```

```

        // Update the bias
        *bias += LSLRGetBias(&lslr);

    }

    // Free memory
    VecFree(&beta);
    VecFree(&Y);

}

// If the user requested the bias
if (bias != NULL) {

    // Complete the calculation of the bias
    *bias /= (float)dimOutputs;

}

// Free memory
LeastSquareLinRegFreeStatic(&lslr);
MatFree(&X);

// Return the result
return res;

}

// Calculate the weights of the control points of the BBody 'that'
// for the given 'inputs'
// Return a VecFloat with weights in same order as 'that->_ctrl'
VecFloat* BBodyGetWeightCtrlPt(
    const BBody* that,
    const VecFloat* inputs) {

#ifdef BUILDMODE == 0

    if (that == NULL) {

        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(
            BCurveErr->_msg,
            "'that' is null");
        PBErrCatch(BCurveErr);

    }

    if (inputs == NULL) {

        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(
            BCurveErr->_msg,
            "'inputs' is null");
        PBErrCatch(BCurveErr);

    }

    long dimInputs =
        VecGet(
            BBodyDim(that),

```

```

    0);
    if (VecGetDim(inputs) != dimInputs) {

        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(
            BCurveErr->_msg,
            "Invalid inputs dimension (%ld>%ld)",
            VecGetDim(inputs),
            dimInputs);
        PBErrCatch(BCurveErr);

    }

#endif

    // Allocate memory for the result
    VecFloat* weights = VecFloatCreate(BBodyGetNbCtrl(that));

    // Get the number of inputs of BBody
    short nbInputs =
        VecGet(
            BBodyDim(that),
            0);

    // Allocate memory for the weights per input
    VecFloat** weightInputs =
        PBErrMalloc(
            BCurveErr,
            sizeof(VecFloat*) * nbInputs);

    // Create a BCurve of same order as the BBody to calculate the
    // weights per inputs
    // The dimension of outputs doesn't matter here, force it to 1
    BCurve* bcurve =
        BCurveCreate(
            BBodyGetOrder(that),
            1);

    // Get the weights of each input
    for (
        short iInp = nbInputs;
        iInp--;) {

        float t =
            VecGet(
                inputs,
                iInp);
        weightInputs[iInp] =
            BCurveGetWeightCtrlPt(
                bcurve,
                t);

    }

    // Loop on control points of the BBody
    int iCtrl = 0;
    int nbCtrlInput = BCurveGetNbCtrl(bcurve);
    VecShort* step = VecShortCreate(nbInputs);
    VecShort* bounds = VecShortCreate(nbInputs);
    VecSetAll(
        bounds,
        nbCtrlInput);

```

```

bool flagStep = true;
do {

    // Get the weight of the control
    // This is the product of the respective weight per input
    float weight = 1.0;
    for (
        short iInp = nbInputs;
        iInp--;) {

        int jCtrl =
            VecGet(
                step,
                iInp);

        weight *=
            VecGet(
                weightInputs[iInp],
                jCtrl);

    }

    // Set the weight in the result
    VecSet(
        weights,
        iCtrl,
        weight);

    // Step to the next control
    ++iCtrl;
    flagStep =
        VecStep(
            step,
            bounds);

} while (flagStep);

// Free memory
VecFree(&step);
VecFree(&bounds);
BCurveFree(&bcurve);
for (
    short iInp = nbInputs;
    iInp--;) {

    VecFree(weightInputs + iInp);

}
free(weightInputs);

// Return the result
return weights;

}

```

## 3.2 bcurve-inline.c

```
// ===== BCURVE-static inline.C =====
```

```

// ----- BCurve

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

// Get the number of control point of the BCurve 'that'
#if BUILDMODE != 0
static inline
#endif
int BCurveGetNbCtrl(BCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return BCurveGetOrder(that) + 1;
}

// Set the value of the iCtrl-th control point to v
#if BUILDMODE != 0
static inline
#endif
void BCurveSetCtrl(BCurve* const that, const int iCtrl,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (iCtrl < 0 || iCtrl > that->_order) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
            iCtrl, that->_order);
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != BCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'v' 's dimension is invalid (%ld<=%d)",
            VecGetDim(v), BCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    // Set the values
    VecCopy(that->_ctrl[iCtrl], v);
}

// Get a copy of the iCtrl-th control point
#if BUILDMODE != 0
static inline
#endif
VecFloat* BCurveGetCtrl(const BCurve* const that, const int iCtrl) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
    }
#endif
}

```



```

        PBErCatch(BCurveErr);
    }
    if (iCtrl < 0 || iCtrl > that->_order) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<%d)",
            iCtrl, that->_order);
        PBErCatch(BCurveErr);
    }
#endif
    // Return a copy of the control point
    return VecClone(that->_ctrl[iCtrl]);
}

// Get the iCtrl-th control point
#if BUILDMODE != 0
static inline
#endif
const VecFloat* BCurveCtrl(const BCurve* const that, const int iCtrl) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErCatch(BCurveErr);
    }
    if (iCtrl < 0 || iCtrl > that->_order) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<%d)",
            iCtrl, that->_order);
        PBErCatch(BCurveErr);
    }
#endif
    // Return the control point
    return that->_ctrl[iCtrl];
}

// Get the 'iDim'-th value of the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
float BCurveCtrlGet(const BCurve* const that, const int iCtrl,
    const int iDim) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErCatch(BCurveErr);
    }
    if (iCtrl < 0 || iCtrl > that->_order) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<%d)",
            iCtrl, that->_order);
        PBErCatch(BCurveErr);
    }
    if (iDim < 0 || iDim > that->_dim) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iDim' is invalid (0<=%d<%d)",
            iDim, that->_dim);
        PBErCatch(BCurveErr);
    }
#endif
    // Return the value
    return VecGet(BCurveCtrl(that, iCtrl), iDim);
}

```

```

}

// Get the order of the BCurve
#if BUILDMODE != 0
static inline
#endif
int BCurveGetOrder(const BCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_order;
}

// Get the dimension of the BCurve
#if BUILDMODE != 0
static inline
#endif
int BCurveGetDim(const BCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_dim;
}

// Get the approximate length of the BCurve (sum of dist between
// control points)
#if BUILDMODE != 0
static inline
#endif
float BCurveGetApproxLen(const BCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Declare a variable to calculate the length
    float res = 0.0;
    // Calculate the length
    for (int iCtrl = that->_order; iCtrl--;)
        res += VecDist(that->_ctrl[iCtrl], that->_ctrl[iCtrl + 1]);
    // Return the length
    return res;
}

// Return the center of the BCurve (average of control points)
#if BUILDMODE != 0
static inline
#endif
VecFloat* BCurveGetCenter(const BCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {

```

```

        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Sum all the control points
    VecFloat* center = VecClone(that->_ctrl[that->_order]);
    for (int iCtrl = that->_order; iCtrl--;)
        VecOp(center, 1.0, that->_ctrl[iCtrl], 1.0);
    // Get the average
    VecScale(center, 1.0 / (float)(that->_order + 1));
    // Return the result
    return center;
}

// Rotate the curve CCW by 'theta' radians relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void BCurveRotOrigin(BCurve* const that, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (that->_dim != 2) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=2)",
                    that->_dim);
            PBErrCatch(BCurveErr);
        }
    #endif
    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;)
        // Rotate the control point
        VecRot(that->_ctrl[iCtrl], theta);
}

// Rotate the curve CCW by 'theta' radians relatively to its
// first control point
#if BUILDMODE != 0
static inline
#endif
void BCurveRotStart(BCurve* const that, const float theta) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (that->_dim != 2) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=2)",
                    that->_dim);
            PBErrCatch(BCurveErr);
        }
    #endif
    // For each control point except the first one
    for (int iCtrl = that->_order + 1; iCtrl-- && iCtrl != 0;) {
        // Translate the control point

```

```

        VecOp(that->_ctrl[iCtrl], 1.0, that->_ctrl[0], -1.0);
        // Rotate the control point
        VecRot(that->_ctrl[iCtrl], theta);
        // Translate back the control point
        VecOp(that->_ctrl[iCtrl], 1.0, that->_ctrl[0], 1.0);
    }
}

// Rotate the curve CCW by 'theta' radians relatively to its
// center
#if BUILDMODE != 0
static inline
#endif
void BCurveRotCenter(BCurve* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (that->_dim != 2) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=2)",
            that->_dim);
        PBErrCatch(BCurveErr);
    }
#endif
    // Get the center
    VecFloat* center = BCurveGetCenter(that);
    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;) {
        // Translate the control point
        VecOp(that->_ctrl[iCtrl], 1.0, center, -1.0);
        // Rotate the control point
        VecRot(that->_ctrl[iCtrl], theta);
        // Translate back the control point
        VecOp(that->_ctrl[iCtrl], 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Scale the curve by 'v' relatively to the origin
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleOriginVector(BCurve* const that,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != BCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",

```

```

        VecGetDim(v), BCurveGetDim(that));
    PBErrCatch(BCurveErr);
}
#endif
// For each control point
for (int iCtrl = that->_order + 1; iCtrl--;) {
    VecFloat* ctrl = that->_ctrl[iCtrl];
    // Scale the control point
    for (long dim = 0; dim < VecGetDim(ctrl); ++dim)
        VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
}
}

// Scale the curve by 'c' relatively to the origin
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleOriginScalar(BCurve* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;)
        // Scale the control point
        VecScale(that->_ctrl[iCtrl], c);
}

// Scale the curve by 'v' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleStartVector(BCurve* const that,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != BCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), BCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point except the first one
    for (int iCtrl = that->_order + 1; iCtrl-- && iCtrl != 0;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, that->_ctrl[0], -1.0);
        // Scale the control point

```

```

        for (long dim = 0; dim < VecGetDim(that->_ctrl[iCtrl]); ++dim)
            VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
        // Translate back the control point
        VecOp(ctrl, 1.0, that->_ctrl[0], 1.0);
    }
}

// Scale the curve by 'c' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleStartScalar(BCurve* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point except the first one
    for (int iCtrl = that->_order + 1; iCtrl-- && iCtrl != 0;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, that->_ctrl[0], -1.0);
        // Scale the control point
        VecScale(ctrl, c);
        // Translate back the control point
        VecOp(ctrl, 1.0, that->_ctrl[0], 1.0);
    }
}

// Scale the curve by 'v' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleCenterVector(BCurve* const that,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != BCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), BCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = BCurveGetCenter(that);
    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point

```

```

    VecOp(ctrl, 1.0, center, -1.0);
    // Scale the control point
    for (long dim = 0; dim < VecGetDim(that->_ctrl[iCtrl]); ++dim)
        VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
    // Translate back the control point
    VecOp(ctrl, 1.0, center, 1.0);
}
// Free memory
VecFree(&center);
}

// Scale the curve by 'c' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BCurveScaleCenterScalar(BCurve* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = BCurveGetCenter(that);
    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Scale the control point
        VecScale(ctrl, c);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Translate the curve by 'v'
#if BUILDMODE != 0
static inline
#endif
void _BCurveTranslate(BCurve* const that, const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != BCurveGetDim(that)) {
        BCurveErr->_type = PErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), BCurveGetDim(that));
        PErrCatch(BCurveErr);
    }
#endif
}

```

```

    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;)
        // Translate the control point
        VecOp(that->_ctrl[iCtrl], 1.0, v, 1.0);
}

// ----- SCurve

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

// Get the number of BCurve in the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetNbSeg(const SCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_nbSeg;
}

// Get the dimension of the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetDim(const SCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_dim;
}

// Get the order of the SCurve
#if BUILDMODE != 0
static inline
#endif
int SCurveGetOrder(const SCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_order;
}

// Get a clone of the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveGetCtrl(const SCurve* const that, const int iCtrl) {
#if BUILDMODE == 0

```



```

    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (iCtrl < 0 || iCtrl >= SCurveGetNbCtrl(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
            iCtrl, SCurveGetNbCtrl(that));
        PBErrCatch(BCurveErr);
    }
}
#endif
return VecClone(GSetGet(&(that->_ctrl), iCtrl));
}

// Get the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
const VecFloat* SCurveCtrl(const SCurve* const that, const int iCtrl) {
    if (BUILDMODE == 0)
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (iCtrl < 0 || iCtrl >= SCurveGetNbCtrl(that)) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
                iCtrl, SCurveGetNbCtrl(that));
            PBErrCatch(BCurveErr);
        }
    }
    return (const VecFloat*)GSetGet(&(that->_ctrl), iCtrl);
}

// Get the 'iDim'-th value of the 'iCtrl'-th control point
#if BUILDMODE != 0
static inline
#endif
float SCurveCtrlGet(const SCurve* const that, const int iCtrl,
    const int iDim) {
    if (BUILDMODE == 0)
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (iCtrl < 0 || iCtrl >= SCurveGetNbCtrl(that)) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
                iCtrl, SCurveGetNbCtrl(that) - 1);
            PBErrCatch(BCurveErr);
        }
    }
    if (iDim < 0 || iDim > that->_dim) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iDim' is invalid (0<=%d<=%d)",
            iDim, that->_dim);
        PBErrCatch(BCurveErr);
    }
}
#endif
// Return the value

```

```

    return VecGet(SCurveCtrl(that, iCtrl), iDim);
}

// Get the set of control point of the SCurve 'that'
#if BUILDMODE != 0
static inline
#endif
const GSetVecFloat* SCurveCtrls(const SCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
#endif
    return &(that->_ctrl);
}

// Get a clone of the 'iSeg'-th segment
#if BUILDMODE != 0
static inline
#endif
BCurve* SCurveGetSeg(const SCurve* const that, const int iSeg) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (iSeg < 0 || iSeg >= that->_nbSeg) {
        BCurveErr->_type = PErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iSeg' is invalid (0<=%d<%d)",
            iSeg, that->_nbSeg);
        PErrCatch(BCurveErr);
    }
#endif
    return BCurveClone((BCurve*)GSetGet(&(that->_seg), iSeg));
}

// Get the 'iSeg'-th segment
#if BUILDMODE != 0
static inline
#endif
const BCurve* SCurveSeg(const SCurve* const that, const int iSeg) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (iSeg < 0 || iSeg >= that->_nbSeg) {
        BCurveErr->_type = PErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iSeg' is invalid (0<=%d<%d)",
            iSeg, that->_nbSeg);
        PErrCatch(BCurveErr);
    }
#endif
    return (BCurve*)GSetGet(&(that->_seg), iSeg);
}

// Get the GSet of segments of the SCurve 'that'
#if BUILDMODE != 0

```

```

static inline
#endif
const GSetBCurve* SCurveSegs(const SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return &(that->_seg);
}

// Return the center of the SCurve (average of control points)
#ifdef BUILDMODE != 0
static inline
#endif
VecFloat* SCurveGetCenter(const SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Sum all the control points
    VecFloat* center = VecFloatCreate(that->_dim);
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    do {
        VecOp(center, 1.0, (VecFloat*)GSetIterGet(&iter), 1.0);
    } while (GSetIterStep(&iter));
    // Get the average
    VecScale(center, 1.0 / (float)GSetNbElem(&(that->_ctrl)));
    // Return the result
    return center;
}

// Return the max value for the parameter 'u' of SCurveGet
#ifdef BUILDMODE != 0
static inline
#endif
float SCurveGetMaxU(const SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return (float)(that->_nbSeg);
}

// Get the number of control point in the SCurve
#ifdef BUILDMODE != 0
static inline
#endif
int SCurveGetNbCtrl(const SCurve* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
}

```

```

    }
#endif
    return that->_nbSeg * that->_order + 1;
}

// Rotate the curve CCW by 'theta' radians relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void SCurveRotOrigin(SCurve* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    do {
        // Rotate the control point
        VecRot((VecFloat*)GSetIterGet(&iter), theta);
    } while (GSetIterStep(&iter));
}

// Rotate the curve CCW by 'theta' radians relatively to its
// first control point
#if BUILDMODE != 0
static inline
#endif
void SCurveRotStart(SCurve* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* origin = GSetHead(&(that->_ctrl));
    // For each control point except the first one
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    if (GSetIterStep(&iter)) {
        do {
            VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
            // Translate the control point
            VecOp(ctrl, 1.0, origin, -1.0);
            // Rotate the control point
            VecRot(ctrl, theta);
            // Translate back the control point
            VecOp(ctrl, 1.0, origin, 1.0);
        } while (GSetIterStep(&iter));
    }
}

// Rotate the curve CCW by 'theta' radians relatively to its
// center
#if BUILDMODE != 0
static inline
#endif
void SCurveRotCenter(SCurve* const that, const float theta) {
#if BUILDMODE == 0

```

```

    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // Get the center
    VecFloat* center = SCurveGetCenter(that);
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(amp;that->_ctrl));
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Rotate the control point
        VecRot(ctrl, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    } while (GSetIterStep(&iter));
    // Free memory
    VecFree(&center);
}

// Scale the curve by 'v' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleOriginVector(SCurve* const that,
    const VecFloat* const v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(amp;that->_ctrl));
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        // Scale the control point
        for (int iDim = SCurveGetDim(that); iDim--;)
            VecSet(ctrl, iDim, VecGet(ctrl, iDim) * VecGet(v, iDim));
    } while (GSetIterStep(&iter));
}

// Scale the curve by 'c' relatively to the origin
// of the coordinates system
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleOriginScalar(SCurve* const that, const float c) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(amp;that->_ctrl));

```

```

do {
    // Scale the control point
    VecScale((VecFloat*)GSetIterGet(&iter), c);
} while (GSetIterStep(&iter));
}

// Scale the curve by 'v' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleStartVector(SCurve* const that,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != SCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' 's dimension is invalid (%ld=%d)",
            VecGetDim(v), SCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* origin = GSetHead(&(that->_ctrl));
    // For each control point except the first one
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    if (GSetIterStep(&iter)) {
        do {
            VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
            // Translate the control point
            VecOp(ctrl, 1.0, origin, -1.0);
            // Scale the control point
            for (int iDim = SCurveGetDim(that); iDim--;)
                VecSet(ctrl, iDim, VecGet(ctrl, iDim) * VecGet(v, iDim));
            // Translate back the control point
            VecOp(ctrl, 1.0, origin, 1.0);
        } while (GSetIterStep(&iter));
    }
}

// Scale the curve by 'c' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleStartScalar(SCurve* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* origin = GSetHead(&(that->_ctrl));

```

```

// For each control point except teh first one
GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
if (GSetIterStep(&iter)) {
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        // Translate the control point
        VecOp(ctrl, 1.0, origin, -1.0);
        // Scale the control point
        VecScale(ctrl, c);
        // Translate back the control point
        VecOp(ctrl, 1.0, origin, 1.0);
    } while (GSetIterStep(&iter));
}
}

// Scale the curve by 'v' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _SCurveScaleCenterVector(SCurve* const that,
    const VecFloat* const v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (v == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'v' is null");
            PBErrCatch(BCurveErr);
        }
        if (VecGetDim(v) != SCurveGetDim(that)) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'v' 's dimension is invalid (%ld=%d)",
                VecGetDim(v), SCurveGetDim(that));
            PBErrCatch(BCurveErr);
        }
    #endif
    VecFloat* center = SCurveGetCenter(that);
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Scale the control point
        for (int iDim = SCurveGetDim(that); iDim--;)
            VecSet(ctrl, iDim, VecGet(ctrl, iDim) * VecGet(v, iDim));
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    } while (GSetIterStep(&iter));
    // Free memory
    VecFree(&center);
}

// Scale the curve by 'c' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif

```

```

void _SCurveScaleCenterScalar(SCurve* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = SCurveGetCenter(that);
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    do {
        VecFloat* ctrl = (VecFloat*)GSetIterGet(&iter);
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Scale the control point
        VecScale(ctrl, c);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    } while (GSetIterStep(&iter));
    // Free memory
    VecFree(&center);
}

// Translate the curve by 'v'
#if BUILDMODE != 0
static inline
#endif
void _SCurveTranslate(SCurve* const that, const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != SCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), SCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    // Translate all the control points
    GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
    do {
        VecOp((VecFloat*)GSetIterGet(&iter), 1.0, v, 1.0);
    } while (GSetIterStep(&iter));
}

// Get the value of the SCurve at paramater 'u'
// The value is equal to the value of the floor(u)-th segment at
// value (u - floor(u))
// u can extend beyond [0.0, _nbSeg]
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveGet(const SCurve* const that, const float u) {

```



```

#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
// Declare a variable to memorize the relevant segment
int iSeg = 0;
// Declare a variable to memorize the local parameter value
float uloc = u;
// Get the segment the corresponding to 'u'
if (u < 0.0) {
    iSeg = 0;
} else if (u >= that->_nbSeg) {
    iSeg = that->_nbSeg - 1;
    uloc = u - (float)(that->_nbSeg - 1);
} else {
    iSeg = (int)floor(u);
    uloc = u - (float)iSeg;
}
// Get the value of the BCurve
return BCurveGet(SCurveSeg(that, iSeg), uloc);
}

// Get the approximate length of the SCurve (sum of approxLen
// of its BCurves)
#if BUILDMODE != 0
static inline
#endif
float SCurveGetApproxLen(const SCurve* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
// Declare a variable to memorize the length
float length = 0.0;
// For each segment
GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_seg));
do {
    // Add the length of this segment
    length += BCurveGetApproxLen((BCurve*)GSetIterGet(&iter));
} while (GSetIterStep(&iter));
// Return the result
return length;
}

// Set the 'iCtrl'-th control point to 'v'
#if BUILDMODE != 0
static inline
#endif
void SCurveSetCtrl(SCurve* const that, const int iCtrl,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
}

```

```

    if (v == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErCatch(BCurveErr);
    }
    if (iCtrl < 0 || iCtrl >= SCurveGetNbCtrl(that)) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
            iCtrl, SCurveGetNbCtrl(that));
        PBErCatch(BCurveErr);
    }
}
#endif
VecCopy((VecFloat*)GSetGet(&(that->_ctrl), iCtrl), v);
}

// Set the 'iDim'-th value of the 'iCtrl'-th control point to 'v'
#if BUILDMODE != 0
static inline
#endif
void SCurveCtrlSet(SCurve* const that, const int iCtrl, const int iDim,
    float v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErCatch(BCurveErr);
        }
        if (iCtrl < 0 || iCtrl >= SCurveGetNbCtrl(that)) {
            BCurveErr->_type = PBErTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
                iCtrl, SCurveGetNbCtrl(that));
            PBErCatch(BCurveErr);
        }
        if (iDim < 0 || iDim >= SCurveGetDim(that)) {
            BCurveErr->_type = PBErTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'iDim' is invalid (0<=%d<=%d)",
                iDim, SCurveGetDim(that));
            PBErCatch(BCurveErr);
        }
    }
    #endif
    VecSet((VecFloat*)GSetGet(&(that->_ctrl), iCtrl), iDim, v);
}

// Create a new SCurve from the outline of the Shapoid 'shap'
// The Shapoid must be of dimension 2
// Control points are ordered CCW of the Shapoid
#if BUILDMODE != 0
static inline
#endif
SCurve* SCurveCreateFromShapoid(const Shapoid* const shap) {
    #if BUILDMODE == 0
        if (shap == NULL) {
            BCurveErr->_type = PBErTypeNullPointer;
            sprintf(BCurveErr->_msg, "'shap' is null");
            PBErCatch(BCurveErr);
        }
        if (ShapoidGetDim(shap) != 2) {
            BCurveErr->_type = PBErTypeInvalidArg;
            sprintf(BCurveErr->_msg,
                "'shap' 's dimension is invalid (%d==2)",
                ShapoidGetDim(shap));
        }
    }
    #endif
}

```

```

        PBErCatch(BCurveErr);
    }
#endif
// Declare the new curve
SCurve* ret = NULL;
// Call the appropriate function according to the type of the Shapoid
switch (ShapoidGetType(shap)) {
    case ShapoidTypeFacoid:
        ret = SCurveCreateFromFacoid((Facoid*)shap);
        break;
    case ShapoidTypePyramidoid:
        ret = SCurveCreateFromPyramidoid((Pyramidoid*)shap);
        break;
    case ShapoidTypeSpheroid:
        ret = SCurveCreateFromSpheroid((Spheroid*)shap);
        break;
    default:
        break;
}
// Return the new curve
return ret;
}

// ----- SCurveIter

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

// Set the attached SCurve of the SCurveIter 'that' to 'curve'
#if BUILDMODE != 0
static inline
#endif
void SCurveIterSetCurve(SCurveIter* const that,
    const SCurve* const curve) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErCatch(BCurveErr);
    }
    if (curve == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'curve' is null");
        PBErCatch(BCurveErr);
    }
}
#endif
    that->_curve = curve;
}

// Set the delta of the SCurveIter 'that' to 'delta'
#if BUILDMODE != 0
static inline
#endif
void SCurveIterSetDelta(SCurveIter* const that, const float delta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErCatch(BCurveErr);
    }
    if (delta <= 0.0) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'delta' is invalid (%f>0)", delta);
    }
}

```

```

        PBErrCatch(BCurveErr);
    }
#endif
    that->_delta = delta;
}

// Get the attached curve of the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
const SCurve* SCurveIterCurve(const SCurveIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_curve;
}

// Get the delta of the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
float SCurveIterGetDelta(const SCurveIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_delta;
}

// Init the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
void SCurveIterInit(SCurveIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    that->_curPos = 0.0;
}

// Step the SCurveIter 'that'
// Return false if it couldn't step, true else
#if BUILDMODE != 0
static inline
#endif
bool SCurveIterStep(SCurveIter* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
}

```

```

    }
#endif
    if (that->_curPos >
        SCurveGetMaxU(SCurveIterCurve(that)) - PBMath_EPSILON)
        return false;
    that->_curPos += that->_delta;
    if (that->_curPos > SCurveGetMaxU(SCurveIterCurve(that)))
        that->_curPos = SCurveGetMaxU(SCurveIterCurve(that));
    return true;
}

// Step back the SCurveIter 'that'
// Return false if it couldn't step, true else
#if BUILDMODE != 0
static inline
#endif
bool SCurveIterStepP(SCurveIter* const that) {
    if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    if (that->_curPos < PBMath_EPSILON)
        return false;
    that->_curPos -= that->_delta;
    if (that->_curPos < 0.0)
        that->_curPos = 0.0;
    return true;
}

// Get the current value of the internal parameter of the
// SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
float SCurveIterGetPos(const SCurveIter* const that) {
    if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    if (that == NULL)
        return that->_curPos;
}

// Get the current value of the attached SCurve at the current
// internal position of the SCurveIter 'that'
#if BUILDMODE != 0
static inline
#endif
VecFloat* SCurveIterGet(const SCurveIter* const that) {
    if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    if (that == NULL)
        return SCurveGet(SCurveIterCurve(that), that->_curPos);
}

```

```

}

// ----- BBody

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

// Set the value of the iCtrl-th control point to v
#if BUILDMODE != 0
static inline
#endif
void _BBodySetCtrl(BBody* const that, const VecShort* const iCtrl,
                  const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (iCtrl == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'iCtrl' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(iCtrl) != VecGet(&(that->_dim), 0)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'iCtrl' is invalid (%ld=%d)",
                VecGetDim(iCtrl), VecGet(&(that->_dim), 0));
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != VecGet(&(that->_dim), 1)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
                VecGetDim(v), VecGet(&(that->_dim), 1));
        PBErrCatch(BCurveErr);
    }
}
#endif
// Get the index of the ctrl
int index = BBodyGetIndexCtrl(that, iCtrl);
// If we could get the index
if (index != -1)
    // Set the ctrl
    VecCopy(that->_ctrl[index], v);
}

// Get the number of control points of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
int BBodyGetNbCtrl(const BBody* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
}
#endif
// Return the number of control points

```

```

    return powi(that->_order + 1, VecGet(&(that->_dim), 0));
}

// Get the the 'iCtrl'-th control point of 'that'
#if BUILDMODE != 0
static inline
#endif
const VecFloat* _BBodyCtrl(const BBody* const that,
    const VecShort* const iCtrl) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (iCtrl == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'iCtrl' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(iCtrl) != VecGet(&(that->_dim), 0)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'iCtrl' is invalid (%ld=%d)",
            VecGetDim(iCtrl), VecGet(&(that->_dim), 0));
        PBErrCatch(BCurveErr);
    }
#endif
    // Get the index
    int index = BBodyGetIndexCtrl(that, iCtrl);
    // If we could get the index
    if (index != -1)
        // Return the control
        return that->_ctrl[index];
    // Else, we couldn't get the index
    else
        // Return NULL
        return NULL;
}

// Get the index in _ctrl of the 'iCtrl' control point of 'that'
#if BUILDMODE != 0
static inline
#endif
int _BBodyGetIndexCtrl(const BBody* const that,
    const VecShort* const iCtrl) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (iCtrl == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'iCtrl' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(iCtrl) != VecGet(&(that->_dim), 0)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'iCtrl' is invalid (%ld=%d)",
            VecGetDim(iCtrl), VecGet(&(that->_dim), 0));
        PBErrCatch(BCurveErr);
    }
#endif
}

```

```

#endif
    for (long iDim = VecGetDim(iCtrl); iDim--;)
        if (VecGet(iCtrl, iDim) < 0 ||
            VecGet(iCtrl, iDim) > that->_order)
            return -1;
    // Declare a variable to memorize the dimension of input
    long dim = VecGetDim(iCtrl);
    // Get the index
    int index = 0;
    for (long iDim = 0; iDim < dim; ++iDim)
        index += index * that->_order + VecGet(iCtrl, iDim);
    // return the index
    return index;
}

// Get the order of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
int BBodyGetOrder(const BBody* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_order;
}

// Get the dimensions of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
const VecShort2D* BBodyDim(const BBody* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return &(that->_dim);
}

// Get a copy of the dimensions of the BBody 'that'
#if BUILDMODE != 0
static inline
#endif
VecShort2D BBodyGetDim(const BBody* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    return that->_dim;
}

// Return the center of the BBody (average of control points)
#if BUILDMODE != 0

```



```

static inline
#endif
VecFloat* BBodyGetCenter(const BBody* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
#endif
    // Sum all the control points
    VecFloat* center = VecFloatCreate(VecGet(BBodyDim(that), 1));
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;)
        VecOp(center, 1.0, that->_ctrl[iCtrl], 1.0);
    // Get the average
    VecScale(center, 1.0 / (float)(BBodyGetNbCtrl(that)));
    // Return the result
    return center;
}

// Translate the BBody by 'v'
#if BUILDMODE != 0
static inline
#endif
void _BBodyTranslate(BBody* const that, const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != VecGet(BBodyDim(that), 1)) {
        BCurveErr->_type = PErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), VecGet(BBodyDim(that), 1));
        PErrCatch(BCurveErr);
    }
#endif
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;)
        // Translate the control point
        VecOp(that->_ctrl[iCtrl], 1.0, v, 1.0);
}

// Scale the BBody by 'v' relatively to the origin
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleOriginVector(BBody* const that,
    const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (v == NULL) {

```

```

        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != VecGet(BBodyDim(that), 1)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Scale the control point
        for (long dim = 0; dim < VecGetDim(ctrl); ++dim)
            VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
    }
}

// Scale the BBody by 'c' relatively to the origin
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleOriginScalar(BBody* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;)
        // Scale the control point
        VecScale(that->_ctrl[iCtrl], c);
}

// Scale the BBody by 'v' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleStartVector(BBody* const that, const VecFloat* const v) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (v == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGetDim(v) != VecGet(BBodyDim(that), 1)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
            VecGetDim(v), VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
}
#endif

```

```

// For each control point except the first one
for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
    VecFloat* ctrl = that->_ctrl[iCtrl];
    // Translate the control point
    VecOp(ctrl, 1.0, that->_ctrl[0], -1.0);
    // Scale the control point
    for (long dim = 0; dim < VecGetDim(that->_ctrl[iCtrl]); ++dim)
        VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
    // Translate back the control point
    VecOp(ctrl, 1.0, that->_ctrl[0], 1.0);
}
}

// Scale the BBody by 'c' relatively to its origin
// (first control point)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleStartScalar(BBody* const that, const float c) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
    #endif
    // For each control point except the first one
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, that->_ctrl[0], -1.0);
        // Scale the control point
        VecScale(ctrl, c);
        // Translate back the control point
        VecOp(ctrl, 1.0, that->_ctrl[0], 1.0);
    }
}

// Scale the BBody by 'v' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleCenterVector(BBody* const that,
    const VecFloat* const v) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErrCatch(BCurveErr);
        }
        if (v == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'v' is null");
            PBErrCatch(BCurveErr);
        }
        if (VecGetDim(v) != VecGet(BBodyDim(that), 1)) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "Dimension of 'v' is invalid (%ld=%d)",
                VecGetDim(v), VecGet(BBodyDim(that), 1));
            PBErrCatch(BCurveErr);
        }
    #endif
}

```

```

#endif
VecFloat* center = BBodyGetCenter(that);
// For each control point
for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
    VecFloat* ctrl = that->_ctrl[iCtrl];
    // Translate the control point
    VecOp(ctrl, 1.0, center, -1.0);
    // Scale the control point
    for (long dim = 0; dim < VecGetDim(that->_ctrl[iCtrl]); ++dim)
        VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
    // Translate back the control point
    VecOp(ctrl, 1.0, center, 1.0);
}
// Free memory
VecFree(&center);
}

// Scale the BBody by 'c' relatively to its center
// (average of control points)
#if BUILDMODE != 0
static inline
#endif
void _BBodyScaleCenterScalar(BBody* const that, const float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = BBodyGetCenter(that);
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Scale the control point
        VecScale(ctrl, c);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around 'axis'
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotAxisOrigin(BBody* const that,
    const VecFloat3D* const axis, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (axis == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'axis' is null");
    }
#endif
}

```

```

    PBErCatch(BCurveErr);
}
if (VecGet(BBodyDim(that), 1) != 3) {
    BCurveErr->_type = PBErTypeNullPointer;
    sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
        VecGet(BBodyDim(that), 1));
    PBErCatch(BCurveErr);
}
#endif
// For each control point
for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
    VecFloat* ctrl = that->_ctrl[iCtrl];
    // Rotate the control point
    VecRotAxis((VecFloat3D*)ctrl, axis, theta);
}
}

// Rotate the BBody by 'theta' relatively to the center
// of the body around 'axis'
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotAxisCenter(BBody* const that,
    const VecFloat3D* const axis, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErCatch(BCurveErr);
    }
    if (axis == NULL) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'axis' is null");
        PBErCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErCatch(BCurveErr);
    }
#endif
    VecFloat* center = BBodyGetCenter(that);
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Rotate the control point
        VecRotAxis((VecFloat3D*)ctrl, axis, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around 'axis'
// dim[1] of BBody must be 3
#if BUILDMODE != 0

```

```

static inline
#endif
void BBodyRotAxisStart(BBody* const that, const VecFloat3D* const axis,
    const float theta) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (axis == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'axis' is null");
        PErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PErrCatch(BCurveErr);
    }
#endif
    VecFloat* start = that->_ctrl[0];
    // For each control point except the first one
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, start, -1.0);
        // Rotate the control point
        VecRotAxis((VecFloat3D*)ctrl, axis, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, start, 1.0);
    }
}

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around X
// dim[1] of BBody must be 3
#ifdef BUILDMODE != 0
static inline
#endif
void BBodyRotXOrigin(BBody* const that, const float theta) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PErrCatch(BCurveErr);
    }
#endif
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Rotate the control point
        VecRotX((VecFloat3D*)ctrl, theta);
    }
}

```

```

// Rotate the BBody by 'theta' relatively to the center
// of the body around X
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotXCenter(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = BBodyGetCenter(that);
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Rotate the control point
        VecRotX((VecFloat3D*)ctrl, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around X
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotXStart(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PErrCatch(BCurveErr);
    }
#endif
    VecFloat* start = that->_ctrl[0];
    // For each control point except the first one
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, start, -1.0);
    }
}

```

```

        // Rotate the control point
        VecRotX((VecFloat3D*)ctrl, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, start, 1.0);
    }
}

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around Y
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotYOrigin(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Rotate the control point
        VecRotY((VecFloat3D*)ctrl, theta);
    }
}

// Rotate the BBody by 'theta' relatively to the center
// of the body around Y
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotYCenter(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = BBodyGetCenter(that);
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Rotate the control point

```



```

        VecRotY((VecFloat3D*)ctrl, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around Y
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotYStart(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* start = that->_ctrl[0];
    // For each control point except the first one
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, start, -1.0);
        // Rotate the control point
        VecRotY((VecFloat3D*)ctrl, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, start, 1.0);
    }
}

// Rotate the BBody by 'theta' relatively to the origin
// of the coordinates system around Z
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotZOrigin(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
    // For each control point

```

```

    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Rotate the control point
        VecRotZ((VecFloat3D*)ctrl, theta);
    }
}

// Rotate the BBody by 'theta' relatively to the center
// of the body around Z
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotZCenter(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat* center = BBodyGetCenter(that);
    // For each control point
    for (int iCtrl = BBodyGetNbCtrl(that); iCtrl--;) {
        VecFloat* ctrl = that->_ctrl[iCtrl];
        // Translate the control point
        VecOp(ctrl, 1.0, center, -1.0);
        // Rotate the control point
        VecRotZ((VecFloat3D*)ctrl, theta);
        // Translate back the control point
        VecOp(ctrl, 1.0, center, 1.0);
    }
    // Free memory
    VecFree(&center);
}

// Rotate the BBody by 'theta' relatively to the first control point
// of the body around Z
// dim[1] of BBody must be 3
#if BUILDMODE != 0
static inline
#endif
void BBodyRotZStart(BBody* const that, const float theta) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (VecGet(BBodyDim(that), 1) != 3) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 's dimension is invalid (%d=3)",
            VecGet(BBodyDim(that), 1));
        PBErrCatch(BCurveErr);
    }
#endif
}

```

```

VecFloat* start = that->_ctrl[0];
// For each control point except the first one
for (int iCtrl = BBodyGetNbCtrl(that); iCtrl-- && iCtrl != 0;) {
    VecFloat* ctrl = that->_ctrl[iCtrl];
    // Translate the control point
    VecOp(ctrl, 1.0, start, -1.0);
    // Rotate the control point
    VecRotZ((VecFloat3D*)ctrl, theta);
    // Translate back the control point
    VecOp(ctrl, 1.0, start, 1.0);
}
}

```

## 4 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=pbmath
$($(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)/

```

## 5 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"

```

```

#include "bcurve.h"

#define RANDOMSEED 0

void UnitTestBCurveCreateCloneFree() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    if (curve->_dim != dim || curve->_order != order){
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveCreate failed");
        PBErrCatch(BCurveErr);
    }
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        if (VecIsEqual(curve->_ctrl[iCtrl], v) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveCreate failed");
            PBErrCatch(BCurveErr);
        }
    }
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }
    BCurve* clone = BCurveClone(curve);
    if (clone->_dim != dim || clone->_order != order){
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveClone failed");
        PBErrCatch(BCurveErr);
    }
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        if (VecIsEqual(clone->_ctrl[iCtrl], v) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveClone failed");
            PBErrCatch(BCurveErr);
        }
    }
    BCurveFree(&curve);
    if (curve != NULL) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveFree failed");
        PBErrCatch(BCurveErr);
    }
    BCurveFree(&clone);
    VecFree(&v);
    printf("UnitTestBCurveCreateCloneFree OK\n");
}

void UnitTestBCurveLoadSavePrint() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }
}

```

```

BCurvePrint(curve, stdout);
printf("\n");
FILE* file = fopen("./bcurve.txt", "w");
if (BCurveSave(curve, file, false) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BCurveSave failed");
    PBErrCatch(BCurveErr);
}
BCurve* load = BCurveCreate(order, dim);
fclose(file);
file = fopen("./bcurve.txt", "r");
if (BCurveLoad(&load, file) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BCurveLoad failed");
    PBErrCatch(BCurveErr);
}
fclose(file);
if (load->_dim != dim || load->_order != order) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BCurveLoad failed");
    PBErrCatch(BCurveErr);
}
for (int iCtrl = order + 1; iCtrl--;) {
    for (int iDim = dim; iDim--;)
        VecSet(v, iDim, iCtrl * dim + iDim);
    if (VecIsEqual(load->_ctrl[iCtrl], v) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveLoad failed");
        PBErrCatch(BCurveErr);
    }
}
BCurveFree(&curve);
BCurveFree(&load);
VecFree(&v);
printf("UnitTestBCurveLoadSavePrint OK\n");
}

void UnitTestBCurveGetSetCtrl() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
        if (VecIsEqual(curve->_ctrl[iCtrl], v) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveSetCtrl failed");
            PBErrCatch(BCurveErr);
        }
    }
    VecFloat* w = BCurveGetCtrl(curve, iCtrl);
    if (VecIsEqual(w, v) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveGetCtrl failed");
        PBErrCatch(BCurveErr);
    }
}
VecFree(&w);
if (VecIsEqual(BCurveCtrl(curve, iCtrl), v) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BCurveCtrl failed");
    PBErrCatch(BCurveErr);
}

```

```

    }
}
BCurveFree(&curve);
VecFree(&v);
printf("UnitTestBCurveGetSetCtrl OK\n");
}

void UnitTestBCurveGet() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }
    for (float u = 0.0; u < 1.0 + PBMath_EPSILON; u += 0.1) {
        VecFloat* w = BCurveGet(curve, u);
        if (ISEQUALF(VecGet(w, 0), u * 6.0) == false ||
            ISEQUALF(VecGet(w, 1), u * 6.0 + 1.0) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveGet failed");
            PBErrCatch(BCurveErr);
        }
        VecFree(&w);
    }
    BCurveFree(&curve);
    VecFree(&v);
    printf("UnitTestBCurveGet OK\n");
}

void UnitTestBCurveGetOrderDimNbCtrl() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    if (BCurveGetOrder(curve) != order) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveGetOrder failed");
        PBErrCatch(BCurveErr);
    }
    if (BCurveGetDim(curve) != dim) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveGetDim failed");
        PBErrCatch(BCurveErr);
    }
    if (BCurveGetNbCtrl(curve) != order + 1) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveGetNbCtrl failed");
        PBErrCatch(BCurveErr);
    }
    BCurveFree(&curve);
    printf("UnitTestBCurveGetOrderDimNbCtrl OK\n");
}

void UnitTestBCurveGetApproxLenCenter() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)

```

```

        VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }
    float len = BCurveGetApproxLen(curve);
    if (ISEQUALF(len, 8.485281) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveGetApproxLen failed");
        PBErrCatch(BCurveErr);
    }
    VecFloat* center = BCurveGetCenter(curve);
    VecSet(v, 0, 3.0);
    VecSet(v, 1, 4.0);
    if (VecIsEqual(v, center) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveGetCenter failed");
        PBErrCatch(BCurveErr);
    }
    VecFree(&center);
    BCurveFree(&curve);
    VecFree(&v);
    printf("UnitTestBCurveGetApproxLenCenter OK\n");
}

void UnitTestBCurveRot() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }
    float theta = PBMath_HALFPI;
    BCurveRotOrigin(curve, theta);
    float pa[8] = {-1.0, 0.0, -3.0, 2.0, -5.0, 4.0, -7.0, 6.0};
    for (int iCtrl = order + 1; iCtrl--;)
        for (int iDim = dim; iDim--;)
            if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
                pa[iCtrl * dim + iDim]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BCurveRotOrigin failed");
                PBErrCatch(BCurveErr);
            }
    BCurveRotStart(curve, theta);
    float pb[8] = {-1.0, 0.0, -3.0, -2.0, -5.0, -4.0, -7.0, -6.0};
    for (int iCtrl = order + 1; iCtrl--;)
        for (int iDim = dim; iDim--;)
            if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
                pb[iCtrl * dim + iDim]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BCurveRotStart failed");
                PBErrCatch(BCurveErr);
            }
    BCurveRotCenter(curve, theta);
    float pc[8] = {-7.0, 0.0, -5.0, -2.0, -3.0, -4.0, -1.0, -6.0};
    for (int iCtrl = order + 1; iCtrl--;)
        for (int iDim = dim; iDim--;)
            if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
                pc[iCtrl * dim + iDim]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BCurveRotCenter failed");
            }
}

```

```

        PBErrCatch(BCurveErr);
    }
    BCurveFree(&curve);
    VecFree(&v);
    printf("UnitTestBCurveRot OK\n");
}

void UnitTestBCurveScale() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }
    float scale = 2.0;
    BCurveScaleOrigin(curve, scale);
    float pa[8] = {0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0};
    for (int iCtrl = order + 1; iCtrl--;)
        for (int iDim = dim; iDim--;)
            if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
                pa[iCtrl * dim + iDim]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BCurveScaleOrigin failed");
                PBErrCatch(BCurveErr);
            }
    BCurveScaleStart(curve, scale);
    float pb[8] = {0.0, 2.0, 8.0, 10.0, 16.0, 18.0, 24.0, 26.0};
    for (int iCtrl = order + 1; iCtrl--;)
        for (int iDim = dim; iDim--;)
            if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
                pb[iCtrl * dim + iDim]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BCurveScaleStart failed");
                PBErrCatch(BCurveErr);
            }
    BCurveScaleCenter(curve, scale);
    float pc[8] = {-12.0, -10.0, 4.0, 6.0, 20.0, 22.0, 36.0, 38.0};
    for (int iCtrl = order + 1; iCtrl--;)
        for (int iDim = dim; iDim--;)
            if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
                pc[iCtrl * dim + iDim]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BCurveScaleCenter failed");
                PBErrCatch(BCurveErr);
            }
    BCurveFree(&curve);
    VecFree(&v);
    printf("UnitTestBCurveScale OK\n");
}

void UnitTestBCurveTranslate() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = order + 1; iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        BCurveSetCtrl(curve, iCtrl, v);
    }

```



```

}
VecSet(v, 0, -1.0);
VecSet(v, 1, -2.0);
BCurveTranslate(curve, v);
for (int iCtrl = order + 1; iCtrl--;) {
    for (int iDim = dim; iDim--;) {
        VecSet(v, iDim, iCtrl * dim + iDim);
        if (ISEQUALF(BCurveCtrlGet(curve, iCtrl, iDim),
            VecGet(v, iDim) - (float)(iDim + 1)) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveTranslate failed");
            PBErrCatch(BCurveErr);
        }
    }
}
BCurveFree(&curve);
VecFree(&v);
printf("UnitTestBCurveTranslate OK\n");
}

void UnitTestBCurveFromCloudPoint() {
    int order = 2;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* vA = VecFloatCreate(dim);
    VecSet(vA, 0, 0.0); VecSet(vA, 1, 0.0);
    BCurveSetCtrl(curve, 0, vA);
    VecFloat* vB = VecFloatCreate(dim);
    VecSet(vB, 0, 0.5); VecSet(vB, 1, 1.0);
    BCurveSetCtrl(curve, 1, vB);
    VecFloat* vC = VecFloatCreate(dim);
    VecSet(vC, 0, 1.0); VecSet(vC, 1, 0.0);
    BCurveSetCtrl(curve, 2, vC);
    GSetVecFloat* set = GSetVecFloatCreate();
    VecFree(&vB);
    vB = BCurveGet(curve, 0.5);
    GSetAppend(set, vA);
    GSetAppend(set, vB);
    GSetAppend(set, vC);
    BCurve* cloud = BCurveFromCloudPoint(set);
    if (cloud == NULL) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BCurveFromCloudPoint failed");
        PBErrCatch(BCurveErr);
    }
    for (float u = 0.0; u < 1.0 + PBMath_EPSILON; u += 0.1) {
        VecFloat* wA = BCurveGet(curve, u);
        VecFloat* wB = BCurveGet(cloud, u);
        if (VecIsEqual(wA, wB) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveFromCloudPoint failed");
            PBErrCatch(BCurveErr);
        }
    }
    VecFree(&wA);
    VecFree(&wB);
}
GSetFree(&set);
BCurveFree(&curve);
BCurveFree(&cloud);
VecFree(&vA);
VecFree(&vB);
VecFree(&vC);

```

```

    printf("UnitTestBCurveFromCloudPoint OK\n");
}

void UnitTestBCurveGetWeightCtrlPt() {
    int order = 2;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* vA = VecFloatCreate(dim);
    VecSet(vA, 0, 0.0); VecSet(vA, 1, 0.0);
    BCurveSetCtrl(curve, 0, vA);
    VecFloat* vB = VecFloatCreate(dim);
    VecSet(vB, 0, 0.5); VecSet(vB, 1, 1.0);
    BCurveSetCtrl(curve, 1, vB);
    VecFloat* vC = VecFloatCreate(dim);
    VecSet(vC, 0, 1.0); VecSet(vC, 1, 0.0);
    BCurveSetCtrl(curve, 2, vC);
    float pa[11] =
        {1.0, 0.81, 0.64, 0.49, 0.36, 0.25, 0.16, 0.09, 0.04, 0.01, 0.0};
    float pb[11] =
        {0.0, 0.18, 0.32, 0.42, 0.48, 0.5, 0.48, 0.42, 0.32, 0.18, 0.0};
    float pc[11] =
        {0.0, 0.01, 0.04, 0.09, 0.16, 0.25, 0.36, 0.49, 0.64, 0.81, 1.0};
    int iArr = 0;
    for (float u = 0.0; u < 1.0 + PBMath_EPSILON; u += 0.1, ++iArr) {
        VecFloat* w = BCurveGetWeightCtrlPt(curve, u);
        if (ISEQUALF(VecGet(w, 0), pa[iArr]) == false ||
            ISEQUALF(VecGet(w, 1), pb[iArr]) == false ||
            ISEQUALF(VecGet(w, 2), pc[iArr]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BCurveGetWeightCtrlPt failed");
            PBErrCatch(BCurveErr);
        }
        VecFree(&w);
    }
    BCurveFree(&curve);
    VecFree(&vA);
    VecFree(&vB);
    VecFree(&vC);
    printf("UnitTestBCurveGetWeightCtrlPt OK\n");
}

void UnitTestBCurveGetBoundingBox() {
    int order = 3;
    int dim = 2;
    BCurve* curve = BCurveCreate(order, dim);
    VecFloat* v = VecFloatCreate(dim);
    VecSet(v, 0, -0.5); VecSet(v, 1, -0.5);
    BCurveSetCtrl(curve, 0, v);
    VecSet(v, 0, 0.0); VecSet(v, 1, 1.0);
    BCurveSetCtrl(curve, 1, v);
    VecSet(v, 0, 1.0); VecSet(v, 1, 1.5);
    BCurveSetCtrl(curve, 2, v);
    VecSet(v, 0, 1.5); VecSet(v, 1, 0.0);
    BCurveSetCtrl(curve, 3, v);
    Facoid* bound = BCurveGetBoundingBox(curve);
    Facoid* check = FacoidCreate(dim);
    float scale = 2.0;
    ShapoidScale(check, scale);
    VecSet(v, 0, -0.5); VecSet(v, 1, -0.5);
    ShapoidTranslate(check, v);
    if (ShapoidIsEqual(bound, check) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
    }
}

```

```

        sprintf(BCurveErr->_msg, "BCurveGetBoundingBox failed");
        PBErCatch(BCurveErr);
    }
    ShapoidFree(&bound);
    ShapoidFree(&check);
    BCurveFree(&curve);
    VecFree(&v);
    printf("UnitTestBCurveGetBoundingBox OK\n");
}

void UnitTestBCurve() {
    UnitTestBCurveCreateCloneFree();
    UnitTestBCurveLoadSavePrint();
    UnitTestBCurveGetSetCtrl();
    UnitTestBCurveGet();
    UnitTestBCurveGetOrderDimNbCtrl();
    UnitTestBCurveGetApproxLenCenter();
    UnitTestBCurveRot();
    UnitTestBCurveScale();
    UnitTestBCurveTranslate();
    UnitTestBCurveFromCloudPoint();
    UnitTestBCurveGetWeightCtrlPt();
    UnitTestBCurveGetBoundingBox();
    printf("UnitTestBCurve OK\n");
}

void UnitTestSCurveCreateCloneFree() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    if (curve->_dim != dim || curve->_order != order ||
        curve->_nbSeg != nbSeg ||
        GSetNbElem(&(curve->_ctrl)) != 1 + order * nbSeg){
        BCurveErr->_type = PBErTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCreate failed");
        PBErCatch(BCurveErr);
    }
    VecFloat* v = VecFloatCreate(dim);
    GSetIterForward iter = GSetIterForwardCreateStatic(&(curve->_ctrl));
    do {
        VecFloat* ctrl = GSetIterGet(&iter);
        if (VecIsEqual(ctrl, v) == false) {
            BCurveErr->_type = PBErTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveCreate failed");
            PBErCatch(BCurveErr);
        }
    } while (GSetIterStep(&iter));
    iter = GSetIterForwardCreateStatic(&(curve->_seg));
    VecFloat* prevCtrl = (VecFloat*)(curve->_ctrl._set._head->_data);
    do {
        BCurve* seg = GSetIterGet(&iter);
        if (seg->_ctrl[0] != prevCtrl) {
            BCurveErr->_type = PBErTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveCreate failed");
            PBErCatch(BCurveErr);
        }
        prevCtrl = seg->_ctrl[order];
    } while (GSetIterStep(&iter));
    iter = GSetIterForwardCreateStatic(&(curve->_ctrl));
    int iCtrl = 0;
    do {

```

```

    VecFloat* ctrl = GSetIterGet(&iter);
    for (int iDim = dim; iDim--;)
        VecSet(ctrl, iDim, iCtrl * dim + iDim);
    ++iCtrl;
} while (GSetIterStep(&iter));
SCurve* clone = SCurveClone(curve);
if (clone->_dim != dim || clone->_order != order ||
    clone->_nbSeg != nbSeg){
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveClone failed");
    PBErrCatch(BCurveErr);
}
iter = GSetIterForwardCreateStatic(&(curve->_ctrl));
GSetIterForward iterClone =
    GSetIterForwardCreateStatic(&(clone->_ctrl));
do {
    VecFloat* ctrl = GSetIterGet(&iter);
    VecFloat* ctrlClone = GSetIterGet(&iterClone);
    if (VecIsEqual(ctrl, ctrlClone) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveClone failed");
        PBErrCatch(BCurveErr);
    }
} while (GSetIterStep(&iter) && GSetIterStep(&iterClone));
SCurveFree(&curve);
if (curve != NULL) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveFree failed");
    PBErrCatch(BCurveErr);
}
SCurveFree(&clone);
VecFree(&v);
printf("UnitTestSCurveCreateCloneFree OK\n");
}

void UnitTestSCurveLoadSavePrint() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    GSetIterForward iter = GSetIterForwardCreateStatic(&(curve->_ctrl));
    int iCtrl = 0;
    do {
        VecFloat* ctrl = GSetIterGet(&iter);
        for (int iDim = dim; iDim--;)
            VecSet(ctrl, iDim, iCtrl * dim + iDim);
        ++iCtrl;
    } while (GSetIterStep(&iter));
    SCurvePrint(curve, stdout);
    printf("\n");
    FILE* file = fopen("./scurve.txt", "w");
    if (SCurveSave(curve, file, false) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveSave failed");
        PBErrCatch(BCurveErr);
    }
    SCurve* load = SCurveCreate(order, dim, nbSeg);
    fclose(file);
    file = fopen("./scurve.txt", "r");
    if (SCurveLoad(&load, file) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveLoad failed");
    }
}

```

```

    PBErCatch(BCurveErr);
}
fclose(file);
if (load->_dim != dim || load->_order != order ||
    load->_order != order) {
    BCurveErr->_type = PBErTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveLoad failed");
    PBErCatch(BCurveErr);
}
iter = GSetIterForwardCreateStatic(&(amp;curve->_ctrl));
GSetIterForward iterLoad =
    GSetIterForwardCreateStatic(&(amp;load->_ctrl));
do {
    VecFloat* ctrl = GSetIterGet(&iter);
    VecFloat* ctrlLoad = GSetIterGet(&iterLoad);
    if (VecIsEqual(ctrl, ctrlLoad) == false) {
        BCurveErr->_type = PBErTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveLoad failed");
        PBErCatch(BCurveErr);
    }
} while (GSetIterStep(&iter) && GSetIterStep(&iterLoad));
SCurveFree(&curve);
SCurveFree(&load);
printf("UnitTestSCurveLoadSavePrint OK\n");
}

void UnitTestSCurveGetSetCtrl() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    VecFloat* v = VecFloatCreate(dim);
    if (SCurveCtrls(curve) != &(curve->_ctrl)) {
        BCurveErr->_type = PBErTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCtrls failed");
        PBErCatch(BCurveErr);
    }
    if (SCurveSegs(curve) != &(curve->_seg)) {
        BCurveErr->_type = PBErTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveSegs failed");
        PBErCatch(BCurveErr);
    }
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        SCurveSetCtrl(curve, iCtrl, v);
    }
    GSetIterForward iter = GSetIterForwardCreateStatic(&(amp;curve->_ctrl));
    int iCtrl = 0;
    do {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        VecFloat* ctrl = GSetIterGet(&iter);
        if (VecIsEqual(ctrl, v) == false) {
            BCurveErr->_type = PBErTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveSetCtrl failed");
            PBErCatch(BCurveErr);
        }
    }
    if (ctrl != SCurveCtrl(curve, iCtrl)) {
        BCurveErr->_type = PBErTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCtrl failed");
        PBErCatch(BCurveErr);
    }
}

```

```

    }
    ctrl = SCurveGetCtrl(curve, iCtrl);
    if (VecIsEqual(ctrl, v) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetCtrl failed");
        PBErrCatch(BCurveErr);
    }
    VecFree(&ctrl);
    ++iCtrl;
} while (GSetIterStep(&iter));
VecFree(&v);
SCurveFree(&curve);
printf("UnitTestSCurveGetSetCtrl OK\n");
}

void UnitTestSCurveGetAddRemoveSeg() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    VecFloat* v = VecFloatCreate(dim);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        SCurveSetCtrl(curve, iCtrl, v);
    }
    for (int iSeg = SCurveGetNbSeg(curve); iSeg--;) {
        BCurve* seg = SCurveGetSeg(curve, iSeg);
        if (BCurveGetDim(seg) != dim || BCurveGetOrder(seg) != order) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveGetSeg failed");
            PBErrCatch(BCurveErr);
        }
        for (int iCtrl = order + 1; iCtrl--;) {
            int jCtrl = iSeg * order + iCtrl;
            if (VecIsEqual(BCurveCtrl(seg, iCtrl),
                SCurveCtrl(curve, jCtrl)) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "SCurveGetSeg failed");
                PBErrCatch(BCurveErr);
            }
            if (BCurveCtrl(SCurveSeg(curve, iSeg), iCtrl) !=
                SCurveCtrl(curve, jCtrl)) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "SCurveSeg failed");
                PBErrCatch(BCurveErr);
            }
        }
    }
    BCurveFree(&seg);
}
SCurveAddSegHead(curve);
SCurveAddSegTail(curve);
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    for (int iDim = dim; iDim--;)
        VecSet(v, iDim, iCtrl * dim + iDim);
    SCurveSetCtrl(curve, iCtrl, v);
}
for (int iSeg = SCurveGetNbSeg(curve); iSeg--;) {
    BCurve* seg = SCurveGetSeg(curve, iSeg);
    if (BCurveGetDim(seg) != dim || BCurveGetOrder(seg) != order) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetSeg failed1");
    }
}

```

```

    PBErrCatch(BCurveErr);
}
for (int iCtrl = order + 1; iCtrl--;) {
    int jCtrl = iSeg * order + iCtrl;
    if (VecIsEqual(BCurveCtrl(seg, iCtrl),
        SCurveCtrl(curve, jCtrl)) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetSeg failed2");
        PBErrCatch(BCurveErr);
    }
    if (BCurveCtrl(SCurveSeg(curve, iSeg), iCtrl) !=
        SCurveCtrl(curve, jCtrl)) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveSeg failed");
        PBErrCatch(BCurveErr);
    }
}
BCurveFree(&seg);
}
SCurveRemoveHeadSeg(curve);
SCurveRemoveTailSeg(curve);
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    for (int iDim = dim; iDim--;)
        VecSet(v, iDim, iCtrl * dim + iDim);
    SCurveSetCtrl(curve, iCtrl, v);
}
for (int iSeg = SCurveGetNbSeg(curve); iSeg--;) {
    BCurve* seg = SCurveGetSeg(curve, iSeg);
    if (BCurveGetDim(seg) != dim || BCurveGetOrder(seg) != order) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetSeg failed");
        PBErrCatch(BCurveErr);
    }
}
for (int iCtrl = order + 1; iCtrl--;) {
    int jCtrl = iSeg * order + iCtrl;
    if (VecIsEqual(BCurveCtrl(seg, iCtrl),
        SCurveCtrl(curve, jCtrl)) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetSeg failed");
        PBErrCatch(BCurveErr);
    }
    if (BCurveCtrl(SCurveSeg(curve, iSeg), iCtrl) !=
        SCurveCtrl(curve, jCtrl)) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveSeg failed");
        PBErrCatch(BCurveErr);
    }
}
}
BCurveFree(&seg);
}
VecFree(&v);
SCurveFree(&curve);
printf("UnitTestSCurveGetAddRemoveSeg OK\n");
}

void UnitTestSCurveGet() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)

```

```

        SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    for (float u = 0.0; u < SCurveGetMaxU(curve) + PBMath_EPSILON;
        u += 0.1) {
        VecFloat* v = SCurveGet(curve, u);
        if (ISEQUALF(VecGet(v, 0), u * 6.0) == false ||
            ISEQUALF(VecGet(v, 1), 1.0 + u * 6.0) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveGet failed");
            PBErrCatch(BCurveErr);
        }
        VecFree(&v);
    }
    SCurveFree(&curve);
    printf("UnitTestSCurveGet OK\n");
}

void UnitTestSCurveGetOrderDimNbSegMaxUNbCtrl() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    if (SCurveGetOrder(curve) != order) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetOrder failed");
        PBErrCatch(BCurveErr);
    }
    if (SCurveGetDim(curve) != dim) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetDim failed");
        PBErrCatch(BCurveErr);
    }
    if (SCurveGetNbSeg(curve) != nbSeg) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetNbSeg failed");
        PBErrCatch(BCurveErr);
    }
    if (ISEQUALF(SCurveGetMaxU(curve), (float)(curve->_nbSeg)) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetMaxU failed");
        PBErrCatch(BCurveErr);
    }
    if (SCurveGetNbCtrl(curve) != nbSeg * order + 1) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetNbCtrl failed");
        PBErrCatch(BCurveErr);
    }
    SCurveFree(&curve);
    printf("UnitTestSCurveGetOrderDimNbSegMaxUNbCtrl OK\n");
}

void UnitTestSCurveGetApproxLenCenter() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)

```



```

        SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    VecFloat* center = SCurveGetCenter(curve);
    VecFloat* check = VecFloatCreate(dim);
    VecSet(check, 0, 9.0);
    VecSet(check, 1, 10.0);
    if (VecIsEqual(center, check) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetCenter failed");
        PBErrCatch(BCurveErr);
    }
    VecFree(&check);
    VecFree(&center);
    float len = 25.455843;
    if (ISEQUALF(SCurveGetApproxLen(curve), len) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetApproxLen failed");
        PBErrCatch(BCurveErr);
    }
    SCurveFree(&curve);
    printf("UnitTestSCurveGetApproxLenCenter OK\n");
}

void UnitTestSCurveRot() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    float theta = PBMATH_HALFPI;
    SCurveRotStart(curve, theta);
    float pa[20] = {0.0, 1.0, -2.0, 3.0, -4.0, 5.0, -6.0, 7.0, -8.0, 9.0,
        -10.0, 11.0, -12.0, 13.0, -14.0, 15.0, -16.0, 17.0, -18.0, 19.0};
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
            pa[iCtrl * 2]) == false ||
            ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
            pa[iCtrl * 2 + 1]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveRotStart failed");
            PBErrCatch(BCurveErr);
        }
    }
    SCurveRotOrigin(curve, theta);
    float pb[20] = {-1.0, 0.0, -3.0, -2.0, -5.0, -4.0, -7.0, -6.0, -9.0,
        -8.0, -11.0, -10.0, -13.0, -12.0, -15.0, -14.0, -17.0, -16.0,
        -19.0, -18.0};
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
            pb[iCtrl * 2]) == false ||
            ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
            pb[iCtrl * 2 + 1]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveRotOrigin failed");
            PBErrCatch(BCurveErr);
        }
    }
    SCurveRotCenter(curve, theta);
    float pc[20] = {-19.0, 0.0, -17.0, -2.0, -15.0, -4.0, -13.0, -6.0,

```

```

        -11.0, -8.0, -9.0, -10.0, -7.0, -12.0, -5.0, -14.0, -3.0, -16.0,
        -1.0, -18.0};
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
        pc[iCtrl * 2]) == false ||
        ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
        pc[iCtrl * 2 + 1]) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveRotCenter failed");
        PBErrCatch(BCurveErr);
    }
}
SCurveFree(&curve);
printf("UnitTestSCurveRot OK\n");
}

void UnitTestSCurveScale() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    float scale = 2.0;
    SCurveScaleStart(curve, scale);
    float pa[20] = {0.0, 1.0, 4.0, 5.0, 8.0, 9.0, 12.0, 13.0, 16.0, 17.0,
        20.0, 21.0, 24.0, 25.0, 28.0, 29.0, 32.0, 33.0, 36.0, 37.0};
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
            pa[iCtrl * 2]) == false ||
            ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
            pa[iCtrl * 2 + 1]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveScaleStart failed");
            PBErrCatch(BCurveErr);
        }
    }
    SCurveScaleOrigin(curve, scale);
    float pb[20] = {0.0, 2.0, 8.0, 10.0, 16.0, 18.0, 24.0, 26.0, 32.0,
        34.0, 40.0, 42.0, 48.0, 50.0, 56.0, 58.0, 64.0, 66.0, 72.0, 74.0};
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
            pb[iCtrl * 2]) == false ||
            ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
            pb[iCtrl * 2 + 1]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveScaleOrigin failed");
            PBErrCatch(BCurveErr);
        }
    }
    SCurveScaleCenter(curve, scale);
    float pc[20] = {-36.0, -34.0, -20.0, -18.0, -4.0, -2.0, 12.0, 14.0,
        28.0, 30.0, 44.0, 46.0, 60.0, 62.0, 76.0, 78.0, 92.0, 94.0,
        108.0, 110.0};
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
            pc[iCtrl * 2]) == false ||
            ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
            pc[iCtrl * 2 + 1]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;

```

```

        sprintf(BCurveErr->_msg, "SCurveScaleCenter failed");
        PBErrCatch(BCurveErr);
    }
}
SCurveFree(&curve);
curve = SCurveCreate(order, dim, nbSeg);
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    for (int iDim = dim; iDim--;)
        SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
}
VecFloat* v = VecFloatCreate(dim);
VecSet(v, 0, 2.0);
VecSet(v, 1, -1.0);
SCurveScaleStart(curve, v);
float pd[20] = {0.0, 1.0, 4.0, -1.0, 8.0, -3.0, 12.0, -5.0, 16.0,
               -7.0, 20.0, -9.0, 24.0, -11.0, 28.0, -13.0, 32.0, -15.0, 36.0,
               -17.0};
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
                pd[iCtrl * 2]) == false ||
        ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
                pd[iCtrl * 2 + 1]) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveScaleStart failed");
        PBErrCatch(BCurveErr);
    }
}
SCurveScaleOrigin(curve, v);
float pe[20] = {0.0, -1.0, 8.0, 1.0, 16.0, 3.0, 24.0, 5.0, 32.0,
               7.0, 40.0, 9.0, 48.0, 11.0, 56.0, 13.0, 64.0, 15.0, 72.0, 17.0};
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
                pe[iCtrl * 2]) == false ||
        ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
                pe[iCtrl * 2 + 1]) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveScaleOrigin failed");
        PBErrCatch(BCurveErr);
    }
}
SCurveScaleCenter(curve, v);
float pf[20] = {-36.0, 17.0, -20.0, 15.0, -4.0, 13.0, 12.0, 11.0,
               28.0, 9.0, 44.0, 7.0, 60.0, 5.0, 76.0, 3.0, 92.0, 1.0, 108.0,
               -1.0};
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
                pf[iCtrl * 2]) == false ||
        ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
                pf[iCtrl * 2 + 1]) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveScaleCenter failed");
        PBErrCatch(BCurveErr);
    }
}
SCurveFree(&curve);
VecFree(&v);
printf("UnitTestSCurveScale OK\n");
}

void UnitTestSCurveTranslate() {
    int order = 3;
    int dim = 2;

```

```

int nbSeg = 3;
SCurve* curve = SCurveCreate(order, dim, nbSeg);
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    for (int iDim = dim; iDim--;)
        SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
}
VecFloat* v = VecFloatCreate(dim);
VecSet(v, 0, -1.0);
VecSet(v, 1, 2.0);
SCurveTranslate(curve, v);
float p[20] = {-1.0, 3.0, 1.0, 5.0, 3.0, 7.0, 5.0, 9.0, 7.0, 11.0,
               9.0, 13.0, 11.0, 15.0, 13.0, 17.0, 15.0, 19.0, 17.0, 21.0};
for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
    if (ISEQUALF(SCurveCtrlGet(curve, iCtrl, 0),
        p[iCtrl * 2]) == false ||
        ISEQUALF(SCurveCtrlGet(curve, iCtrl, 1),
        p[iCtrl * 2 + 1]) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveTranslate failed");
        PBErrCatch(BCurveErr);
    }
}
SCurveFree(&curve);
VecFree(&v);
printf("UnitTestSCurveTranslate OK\n");
}

void UnitTestSCurveGetBoundingBox() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        SCurveCtrlSet(curve, iCtrl, 0,
            cos(PBMATH_QUARTERPI * (float)iCtrl * 0.5));
        SCurveCtrlSet(curve, iCtrl, 1,
            sin(PBMATH_QUARTERPI * (float)iCtrl * 0.5));
    }
    Facoid* bound = SCurveGetBoundingBox(curve);
    if (ISEQUALF(ShapoidPosGet(bound, 0), -1.0) == false ||
        ISEQUALF(ShapoidPosGet(bound, 1), -0.382683) == false ||
        ISEQUALF(ShapoidAxisGet(bound, 0, 0), 2.382684) == false ||
        ISEQUALF(ShapoidAxisGet(bound, 0, 1), 0.0) == false ||
        ISEQUALF(ShapoidAxisGet(bound, 1, 0), 0.0) == false ||
        ISEQUALF(ShapoidAxisGet(bound, 1, 1), 1.765367) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetBoundingBox failed");
        PBErrCatch(BCurveErr);
    }
    ShapoidFree(&bound);
    SCurveFree(&curve);
    printf("UnitTestSCurveGetBoundingBox OK\n");
}

void UnitTestSCurveGetNewDim() {
    int order = 3;
    int dim = 3;
    int nbSeg = 2;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        SCurveCtrlSet(curve, iCtrl, 0, (float)iCtrl);
        SCurveCtrlSet(curve, iCtrl, 1, (float)iCtrl + 1);
    }
}

```

```

    SCurveCtrlSet(curve, iCtrl, 2, (float)iCtrl + 2);
}
SCurve* curveA = SCurveGetNewDim(curve, 2);
if (SCurveGetDim(curveA) != 2 ||
    ISEQUALF(SCurveCtrlGet(curveA, 0, 0), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 0, 1), 1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 1, 0), 1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 1, 1), 2.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 2, 0), 2.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 2, 1), 3.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 3, 0), 3.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 3, 1), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 4, 0), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 4, 1), 5.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 5, 0), 5.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 5, 1), 6.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 6, 0), 6.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveA, 6, 1), 7.0) == false ||
    BCurveCtrl(SCurveSeg(curveA, 0), 0) != SCurveCtrl(curveA, 0) ||
    BCurveCtrl(SCurveSeg(curveA, 0), 1) != SCurveCtrl(curveA, 1) ||
    BCurveCtrl(SCurveSeg(curveA, 0), 2) != SCurveCtrl(curveA, 2) ||
    BCurveCtrl(SCurveSeg(curveA, 0), 3) != SCurveCtrl(curveA, 3) ||
    BCurveCtrl(SCurveSeg(curveA, 1), 0) != SCurveCtrl(curveA, 3) ||
    BCurveCtrl(SCurveSeg(curveA, 1), 1) != SCurveCtrl(curveA, 4) ||
    BCurveCtrl(SCurveSeg(curveA, 1), 2) != SCurveCtrl(curveA, 5) ||
    BCurveCtrl(SCurveSeg(curveA, 1), 3) != SCurveCtrl(curveA, 6)) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveGetNewDim failed");
    PBErrCatch(BCurveErr);
}
SCurve* curveB = SCurveGetNewDim(curve, 4);
if (SCurveGetDim(curveB) != 4 ||
    ISEQUALF(SCurveCtrlGet(curveB, 0, 0), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 0, 1), 1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 0, 2), 2.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 0, 3), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 1, 0), 1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 1, 1), 2.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 1, 2), 3.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 1, 3), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 2, 0), 2.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 2, 1), 3.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 2, 2), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 2, 3), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 3, 0), 3.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 3, 1), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 3, 2), 5.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 3, 3), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 4, 0), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 4, 1), 5.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 4, 2), 6.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 4, 3), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 5, 0), 5.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 5, 1), 6.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 5, 2), 7.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 5, 3), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 6, 0), 6.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 6, 1), 7.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 6, 2), 8.0) == false ||
    ISEQUALF(SCurveCtrlGet(curveB, 6, 3), 0.0) == false ||
    BCurveCtrl(SCurveSeg(curveB, 0), 0) != SCurveCtrl(curveB, 0) ||
    BCurveCtrl(SCurveSeg(curveB, 0), 1) != SCurveCtrl(curveB, 1) ||

```

```

    BCurveCtrl(SCurveSeg(curveB, 0), 2) != SCurveCtrl(curveB, 2) ||
    BCurveCtrl(SCurveSeg(curveB, 0), 3) != SCurveCtrl(curveB, 3) ||
    BCurveCtrl(SCurveSeg(curveB, 1), 0) != SCurveCtrl(curveB, 3) ||
    BCurveCtrl(SCurveSeg(curveB, 1), 1) != SCurveCtrl(curveB, 4) ||
    BCurveCtrl(SCurveSeg(curveB, 1), 2) != SCurveCtrl(curveB, 5) ||
    BCurveCtrl(SCurveSeg(curveB, 1), 3) != SCurveCtrl(curveB, 6)) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveGetNewDim failed");
    PBErrCatch(BCurveErr);
}
SCurveFree(&curve);
SCurveFree(&curveA);
SCurveFree(&curveB);
printf("UnitTestSCurveGetNewDim OK\n");
}

void UnitTestSCurveCreateFromShapoid() {
    Facoid* facoid = FacoidCreate(2);
    Pyramidoid* pyramidoid = PyramidoidCreate(2);
    Spheroid* spheroid = SpheroidCreate(2);
    VecFloat2D v = VecFloatCreateStatic2D();
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0);
    ShapoidSetPos(facoid, &v);
    ShapoidSetPos(pyramidoid, &v);
    ShapoidSetPos(spheroid, &v);
    VecSet(&v, 0, 3.0); VecSet(&v, 1, 4.0);
    ShapoidSetAxis(facoid, 0, &v);
    ShapoidSetAxis(pyramidoid, 0, &v);
    ShapoidSetAxis(spheroid, 0, &v);
    VecSet(&v, 0, -5.0); VecSet(&v, 1, 6.0);
    ShapoidSetAxis(facoid, 1, &v);
    ShapoidSetAxis(pyramidoid, 1, &v);
    ShapoidSetAxis(spheroid, 1, &v);
    SCurve* curve = SCurveCreateFromShapoid((Shapoid*)facoid);
    SCurvePrint(curve, stdout); printf("\n");
    ShapoidPrintln(facoid, stdout);
    if (curve == NULL || SCurveGetDim(curve) != 2 ||
        SCurveGetOrder(curve) != 1 || SCurveGetNbSeg(curve) != 4) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCreateFromFacoid failed");
        PBErrCatch(BCurveErr);
    }
    if (ISEQUALF(SCurveCtrlGet(curve, 0, 0), 1.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 0, 1), 2.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 1, 0), 4.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 1, 1), 6.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 2, 0), -1.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 2, 1), 12.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 3, 0), -4.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 3, 1), 8.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 4, 0), 1.0) == false ||
        ISEQUALF(SCurveCtrlGet(curve, 4, 1), 2.0) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCreateFromFacoid failed");
        PBErrCatch(BCurveErr);
    }
    SCurveFree(&curve);
    curve = SCurveCreateFromShapoid((Shapoid*)pyramidoid);
    if (curve == NULL || SCurveGetDim(curve) != 2 ||
        SCurveGetOrder(curve) != 1 || SCurveGetNbSeg(curve) != 3) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCreateFromPyramidoid failed");
    }
}

```

```

    PBErCatch(BCurveErr);
}
if (ISEQUALF(SCurveCtrlGet(curve, 0, 0), 1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 0, 1), 2.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 1, 0), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 1, 1), 6.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 2, 0), -4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 2, 1), 8.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 3, 0), 1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 3, 1), 2.0) == false) {
    BCurveErr->_type = PBErTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveCreateFromPyramidoid failed");
    PBErCatch(BCurveErr);
}
SCurveFree(&curve);
curve = SCurveCreateFromShapoid((Shapoid*)spheroid);
if (curve == NULL || SCurveGetDim(curve) != 2 ||
    SCurveGetOrder(curve) != 3 || SCurveGetNbSeg(curve) != 4) {
    BCurveErr->_type = PBErTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveCreateFromSpheroid failed");
    PBErCatch(BCurveErr);
}
if (ISEQUALF(SCurveCtrlGet(curve, 0, 0), 2.5) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 0, 1), 4.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 1, 0), 1.119290) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 1, 1), 5.656852) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 2, 0), -0.671574) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 2, 1), 6.104568) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 3, 0), -1.5) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 3, 1), 5.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 4, 0), -2.328426) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 4, 1), 3.895432) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 5, 0), -1.880710) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 5, 1), 1.656852) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 6, 0), -0.5) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 6, 1), 0.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 7, 0), 0.880710) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 7, 1), -1.656852) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 8, 0), 2.671574) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 8, 1), -2.104568) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 9, 0), 3.5) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 9, 1), -1.0) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 10, 0), 4.328426) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 10, 1), 0.104568) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 11, 0), 3.880710) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 11, 1), 2.343148) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 12, 0), 2.5) == false ||
    ISEQUALF(SCurveCtrlGet(curve, 12, 1), 4.0) == false) {
    BCurveErr->_type = PBErTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveCreateFromSpheroid failed");
    PBErCatch(BCurveErr);
}
SCurveFree(&curve);
ShapoidFree(&facoid);
ShapoidFree(&pyramidoid);
ShapoidFree(&spheroid);
printf("UnitTestSCurveCreateFromShapoid OK\n");
}

void UnitTestSCurveGetDistToCurve() {
    int order = 1;
    int dim = 2;

```

```

int nbSeg = 1;
SCurve* curveA = SCurveCreate(order, dim, nbSeg);
SCurve* curveB = SCurveCreate(order, dim, nbSeg);
SCurveCtrlSet(curveA, 0, 0, 0.0);
SCurveCtrlSet(curveA, 0, 1, 0.0);
SCurveCtrlSet(curveA, 1, 0, 1.0);
SCurveCtrlSet(curveA, 1, 1, 0.0);
SCurveCtrlSet(curveB, 0, 0, 0.0);
SCurveCtrlSet(curveB, 0, 1, 2.0);
SCurveCtrlSet(curveB, 1, 0, 1.0);
SCurveCtrlSet(curveB, 1, 1, 2.0);
float dist = SCurveGetDistToCurve(curveA, curveB);
if (ISEQUALF(dist, 2.0) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "SCurveGetDistToCurve failed");
    PBErrCatch(BCurveErr);
}
SCurveFree(&curveA);
SCurveFree(&curveB);
printf("UnitTestSCurveGetDDistToCurve OK\n");
}

void UnitTestSCurveChaikin() {
    int order = 1;
    int dim = 2;
    int nbSeg = 2;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    SCurveCtrlSet(curve, 0, 0, 0.0);
    SCurveCtrlSet(curve, 0, 1, 1.0);
    SCurveCtrlSet(curve, 1, 0, 5.0);
    SCurveCtrlSet(curve, 1, 1, 4.0);
    SCurveCtrlSet(curve, 2, 0, 2.0);
    SCurveCtrlSet(curve, 2, 1, 3.0);
    float strength = 0.1;
    int depth = 2;
    printf("Curve before Chaikin: ");
    SCurvePrint(curve, stdout);printf("\n");
    SCurve* chaikin = SCurveChaikinSubdivision(curve, strength, depth);
    printf("Curve after Chaikin: ");
    SCurvePrint(chaikin, stdout);printf("\n");
    float a[12] = {0.0, 1.0, 4.05, 3.43, 4.52, 3.72, 4.68, 3.88, 4.43, 3.81, 2.0, 3.0};
    for (int i = 0; i < 6; ++i) {
        if (ISEQUALF(SCurveCtrlGet(chaikin, i, 0), a[2 * i]) == false ||
            ISEQUALF(SCurveCtrlGet(chaikin, i, 1), a[2 * i + 1]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveChaikinSubdivision failed");
            PBErrCatch(BCurveErr);
        }
    }
    SCurveFree(&curve);
    SCurveFree(&chaikin);
    printf("UnitTestSCurveChaikin OK\n");
}

void UnitTestSCurve() {
    UnitTestSCurveCreateCloneFree();
    UnitTestSCurveLoadSavePrint();
    UnitTestSCurveGetSetCtrl();
    UnitTestSCurveGetAddRemoveSeg();
    UnitTestSCurveGet();
    UnitTestSCurveGetOrderDimNbSegMaxUNbCtrl();
    UnitTestSCurveGetApproxLenCenter();
}

```



```

    UnitTestSCurveRot();
    UnitTestSCurveScale();
    UnitTestSCurveTranslate();
    UnitTestSCurveGetBoundingBox();
    UnitTestSCurveGetNewDim();
    UnitTestSCurveCreateFromShapoid();
    UnitTestSCurveGetDistToCurve();
    UnitTestSCurveChaikin();
    printf("UnitTestSCurve OK\n");
}

void UnitTestSCurveIterCreate() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    float delta = 0.2;
    SCurveIter iter = SCurveIterCreateStatic(curve, delta);
    if (iter._curve != curve || ISEQUALF(iter._curPos, 0.0) == false ||
        ISEQUALF(iter._delta, delta) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterCreateStatic failed");
        PBErrCatch(BCurveErr);
    }
    SCurveFree(&curve);
    printf("UnitTestSCurveIterCreate OK\n");
}

void UnitTestSCurveIterSetGet() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    float delta = 0.2;
    SCurveIter iter = SCurveIterCreateStatic(curve, delta);
    SCurve* curveB = SCurveCreate(order, dim, nbSeg);
    SCurveIterSetCurve(&iter, curveB);
    if (iter._curve != curveB) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterSetCurve failed");
        PBErrCatch(BCurveErr);
    }
    if (SCurveIterCurve(&iter) != curveB) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterCurve failed");
        PBErrCatch(BCurveErr);
    }
    float deltaB = 0.3;
    SCurveIterSetDelta(&iter, deltaB);
    if (iter._delta != deltaB) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterSetDelta failed");
        PBErrCatch(BCurveErr);
    }
}

```

```

    if (SCurveIterGetDelta(&iter) != deltaB) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterGetDelta failed");
        PBErrCatch(BCurveErr);
    }
    SCurveIterSetCurve(&iter, curve);
    iter._curPos = 0.5;
    if (SCurveIterGetPos(&iter) != 0.5) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterGetPos failed");
        PBErrCatch(BCurveErr);
    }
    VecFloat* pos = SCurveIterGet(&iter);
    if (ISEQUALF(VecGet(pos, 0), 3.0) == false ||
        ISEQUALF(VecGet(pos, 1), 4.0) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterGet failed");
        PBErrCatch(BCurveErr);
    }
    VecFree(&pos);
    SCurveFree(&curve);
    SCurveFree(&curveB);
    printf("UnitTestSCurveIterSetGet OK\n");
}

void UnitTestSCurveIterStep() {
    int order = 3;
    int dim = 2;
    int nbSeg = 3;
    SCurve* curve = SCurveCreate(order, dim, nbSeg);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            SCurveCtrlSet(curve, iCtrl, iDim, iCtrl * dim + iDim);
    }
    float delta = 3.0;
    SCurveIter iter = SCurveIterCreateStatic(curve, delta);
    bool ret = SCurveIterStep(&iter);
    if (ISEQUALF(SCurveIterGetPos(&iter), 3.0) == false ||
        ret == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterStep failed");
        PBErrCatch(BCurveErr);
    }
    ret = SCurveIterStep(&iter);
    if (ISEQUALF(SCurveIterGetPos(&iter), 3.0) == false ||
        ret == true) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterStep failed");
        PBErrCatch(BCurveErr);
    }
    ret = SCurveIterStepP(&iter);
    if (ISEQUALF(SCurveIterGetPos(&iter), 0.0) == false ||
        ret == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterStepP failed");
        PBErrCatch(BCurveErr);
    }
    ret = SCurveIterStepP(&iter);
    if (ISEQUALF(SCurveIterGetPos(&iter), 0.0) == false ||
        ret == true) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveIterStepP failed");
    }
}

```

```

        PBErCatch(BCurveErr);
    }
    SCurveFree(&curve);
    printf("UnitTestSCurveStep OK\n");
}

void UnitTestSCurveIter() {
    UnitTestSCurveIterCreate();
    UnitTestSCurveIterSetGet();
    UnitTestSCurveIterStep();

    printf("UnitTestSCurveIter OK\n");
}

void UnitTestBBodyCreateFree() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    if (VecGet(&(surf->_dim), 0) != VecGet(&dim, 0) ||
        VecGet(&(surf->_dim), 1) != VecGet(&dim, 1) ||
        surf->_order != order) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyCreate failed");
        PBErCatch(BCurveErr);
    }
    BBodyFree(&surf);
    printf("UnitTestBBodyCreateFree OK\n");
}

void UnitTestBBodyGetSet() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    if (BBodyGetOrder(surf) != 1) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyGetOrder failed");
        PBErCatch(BCurveErr);
    }
    if (VecIsEqual(BBodyDim(surf), &dim) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyDim failed");
        PBErCatch(BCurveErr);
    }
    VecShort2D dimB = VecShortCreateStatic2D();
    dimB = BBodyGetDim(surf);
    if (VecIsEqual(&dimB, &dim) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyGetDim failed");
        PBErCatch(BCurveErr);
    }
    if (BBodyGetNbCtrl(surf) != 4) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyGetNbCtrl failed");
        PBErCatch(BCurveErr);
    }
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    if (BBodyGetIndexCtrl(surf, &iCtrl) != 2) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyGetIndexCtrl failed");
    }
}

```

```

    PBErrCatch(BCurveErr);
}
if (BBodyCtrl(surf, &iCtrl) != surf->_ctrl[2]) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BBodyCtrl failed");
    PBErrCatch(BCurveErr);
}
VecFloat3D v = VecFloatCreateStatic3D();
VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 3.0);
BBodySetCtrl(surf, &iCtrl, &v);
if (VecIsEqual(BBodyCtrl(surf, &iCtrl), (VecFloat*)&v) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BBodySetCtrl failed");
    PBErrCatch(BCurveErr);
}
BBodyFree(&surf);
printf("UnitTestBBodyGetSet OK\n");
}

void UnitTestBBodyGet() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecFloat2D u = VecFloatCreateStatic2D();
    float du = 0.2;
    int iCheck = 0;
    float check[75] = {
        0.0,0.0,0.0,0.0,0.2,0.0,0.0,0.4,0.0,0.0,0.6,0.0,0.0,0.8,0.0,
        0.2,0.0,0.0,0.16,0.16,0.04,0.12,0.32,0.08,0.08,0.48,0.12,0.04,
        0.64,0.16,0.4,0.0,0.0,0.32,0.12,0.08,0.24,0.24,0.16,0.16,0.36,
        0.24,0.08,0.48,0.32,0.6,0.0,0.0,0.48,0.08,0.12,0.36,0.16,0.24,
        0.24,0.24,0.36,0.12,0.32,0.48,0.8,0.0,0.0,0.64,0.04,0.16,0.48,
        0.08,0.32,0.32,0.12,0.48,0.16,0.16,0.64
    };
    for (VecSet(&u, 0, 0.0); VecGet(&u, 0) < 1.0;
        VecSet(&u, 0, VecGet(&u, 0) + du)) {
        for (VecSet(&u, 1, 0.0); VecGet(&u, 1) < 1.0;
            VecSet(&u, 1, VecGet(&u, 1) + du)) {
            VecFloat* p = BBodyGet(surf, &u);
            if (ISEQUALF(p->_val[0], check[iCheck]) == false ||
                ISEQUALF(p->_val[1], check[iCheck + 1]) == false ||
                ISEQUALF(p->_val[2], check[iCheck + 2]) == false) {
                BCurveErr->_type = PBErrTypeUnitTestFailed;
                sprintf(BCurveErr->_msg, "BBodyGet failed");
                PBErrCatch(BCurveErr);
            }
            iCheck += 3;
        }
    }
}

```

```

        VecFree(&p);
    }
}
BBodyFree(&surf);
printf("UnitTestBBodyGet OK\n");
}

void UnitTestBBodyClone() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    BBody* clone = BBodyClone(surf);
    if (BBodyGetOrder(clone) != BBodyGetOrder(surf)) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyClone failed");
        PBErrCatch(BCurveErr);
    }
    if (VecIsEqual(BBodyDim(clone), BBodyDim(surf)) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyClone failed");
        PBErrCatch(BCurveErr);
    }
    for (int iCtrl = BBodyGetNbCtrl(clone); iCtrl--;) {
        if (VecIsEqual(clone->_ctrl[iCtrl], surf->_ctrl[iCtrl]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BBodyClone failed");
            PBErrCatch(BCurveErr);
        }
    }
    BBodyFree(&surf);
    BBodyFree(&clone);
    printf("UnitTestBBodyClone OK\n");
}

void UnitTestBBodyPrint() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);

```

```

    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    BBodyPrint(surf, stdout);
    printf("\n");
    BBodyFree(&surf);
    printf("UnitTestBBodyPrint OK\n");
}

void UnitTestBBodyLoadSave() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    FILE* file = fopen("./bbody.txt", "w");
    if (BBodySave(surf, file, false) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodySave failed");
        PBErrCatch(BCurveErr);
    }
    fclose(file);
    BBody* clone = NULL;
    file = fopen("./bbody.txt", "r");
    if (BBodyLoad(&clone, file) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyLoad failed");
        PBErrCatch(BCurveErr);
    }
    fclose(file);
    if (BBodyGetOrder(clone) != BBodyGetOrder(surf)) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyLoadSave failed");
        PBErrCatch(BCurveErr);
    }
    if (VecIsEqual(BBodyDim(clone), BBodyDim(surf)) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyLoadSave failed");
        PBErrCatch(BCurveErr);
    }
    for (int iCtrl = BBodyGetNbCtrl(clone); iCtrl--;) {
        if (VecIsEqual(clone->_ctrl[iCtrl], surf->_ctrl[iCtrl]) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BBodyLoadSave failed");
            PBErrCatch(BCurveErr);
        }
    }
}

```

```

    }
    BBodyFree(&surf);
    BBodyFree(&clone);
    printf("UnitTestBBodyLoadSave OK\n");
}

void UnitTestBBodyGetCenter() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecFloat* center = BBodyGetCenter(surf);
    VecSet(&v, 0, 0.25); VecSet(&v, 1, 0.25); VecSet(&v, 2, 0.25);
    if (VecIsEqual(center, (VecFloat*)&v) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyGetCenter failed");
        PBErrCatch(BCurveErr);
    }
    BBodyFree(&surf);
    VecFree(&center);
    printf("UnitTestBBodyGetCenter OK\n");
}

void UnitTestBBodyTranslate() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 3.0);
    BBodyTranslate(surf, &v);
    float check[12] = {
        1.0, 2.0, 3.0,
        1.0, 3.0, 3.0,
        2.0, 2.0, 3.0,

```

```

1.0,2.0,4.0
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(check[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(check[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(check[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyTranslate failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyFree(&surf);
printf("UnitTestBBodyTranslate OK\n");
}

void UnitTestBBodyScale() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 3.0);
    BBodyScaleCenter(surf, (VecFloat*)&v);
    float checka[12] = {
        0.0,-0.25,-0.5,
        0.0,1.75,-0.5,
        1.0,-0.25,-0.5,
        0.0,-0.25,2.5
    };
    for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
        if (ISEQUALF(checka[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
            false ||
            ISEQUALF(checka[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
            false ||
            ISEQUALF(checka[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
            false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BBodyScaleCenter failed");
            PBErrCatch(BCurveErr);
        }
    }
}
BBodyScaleOrigin(surf, (VecFloat*)&v);
float checkb[12] = {
    0.0,-0.5,-1.5,
    0.0,3.5,-1.5,
    1.0,-0.5,-1.5,

```



```

    0.0,-0.5,7.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkb[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkb[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkb[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyScale failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyScaleStart(surf, (VecFloat*)&v);
float checkc[12] = {
    0.0,-0.5,-1.5,
    0.0,7.5,-1.5,
    1.0,-0.5,-1.5,
    0.0,-0.5,25.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkc[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkc[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkc[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyScale failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyFree(&surf);
printf("UnitTestBBodyScale OK\n");
}

void UnitTestBBodyGetBoundingBox() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    BBodyScaleCenter(surf, (float)2.0);
    Facoid* bound = BBodyGetBoundingBox(surf);
    VecSet(&v, 0, -0.25); VecSet(&v, 1, -0.25); VecSet(&v, 2, -0.25);
    if (VecIsEqual(ShapoidPos(bound), (VecFloat*)&v) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyGetBoundingBox failed");
    }
}

```

```

    PBErrCatch(BCurveErr);
}
VecSet(&v, 0, 2.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
if (VecIsEqual(ShapoidAxis(bound, 0), (VecFloat*)&v) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BBodyGetBoundingBox failed");
    PBErrCatch(BCurveErr);
}
VecSet(&v, 0, 0.0); VecSet(&v, 1, 2.0); VecSet(&v, 2, 0.0);
if (VecIsEqual(ShapoidAxis(bound, 1), (VecFloat*)&v) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BBodyGetBoundingBox failed");
    PBErrCatch(BCurveErr);
}
VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 2.0);
if (VecIsEqual(ShapoidAxis(bound, 2), (VecFloat*)&v) == false) {
    BCurveErr->_type = PBErrTypeUnitTestFailed;
    sprintf(BCurveErr->_msg, "BBodyGetBoundingBox failed");
    PBErrCatch(BCurveErr);
}
ShapoidFree(&bound);
BBodyFree(&surf);
printf("UnitTestBBodyGetBoundingBox OK\n");
}

void UnitTestBBodyRotate() {
    int order = 1;
    VecShort2D dim = VecShortCreateStatic2D();
    VecSet(&dim, 0, 2); VecSet(&dim, 1, 3);
    BBody* surf = BBodyCreate(order, &dim);
    VecShort2D iCtrl = VecShortCreateStatic2D();
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 0);
    VecFloat3D v = VecFloatCreateStatic3D();
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 0);
    VecSet(&v, 0, 1.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 0); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 1.0); VecSet(&v, 2, 0.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    VecSet(&iCtrl, 0, 1); VecSet(&iCtrl, 1, 1);
    VecSet(&v, 0, 0.0); VecSet(&v, 1, 0.0); VecSet(&v, 2, 1.0);
    BBodySetCtrl(surf, &iCtrl, &v);
    float theta = PBMath_HALFPI;
    BBodyRotXCenter(surf, theta);
    float checka[12] = {
        0.0, 0.5, 0.0,
        0.0, 0.5, 1.0,
        1.0, 0.5, 0.0,
        0.0, -0.5, 0.0
    };
    for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
        if (ISEQUALF(checka[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
            false ||
            ISEQUALF(checka[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
            false ||
            ISEQUALF(checka[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
            false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "BBodyRotXCenter failed");
            PBErrCatch(BCurveErr);
        }
    }
}

```

```

    }
}
BBodyRotXOrigin(surf, theta);
float checkb[12] = {
    0.0,0.0,0.5,
    0.0,-1.0,0.5,
    1.0,0.0,0.5,
    0.0,0.0,-0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkb[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkb[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkb[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotXOrigin failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotXStart(surf, theta);
float checkc[12] = {
    0.0,0.0,0.5,
    0.0,0.0,-0.5,
    1.0,0.0,0.5,
    0.0,1.0,0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkc[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkc[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkc[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotXStart failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotYCenter(surf, theta);
float checkd[12] = {
    0.5,0.0,0.5,
    -0.5,0.0,0.5,
    0.5,0.0,-0.5,
    0.5,1.0,0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkd[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkd[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkd[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotYCenter failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotYOrigin(surf, theta);
float checke[12] = {
    0.5,0.0,-0.5,

```

```

    0.5,0.0,0.5,
    -0.5,0.0,-0.5,
    0.5,1.0,-0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checke[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checke[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checke[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotYOrigin failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotYStart(surf, theta);
float checkf[12] = {
    0.5,0.0,-0.5,
    1.5,0.0,-0.5,
    0.5,0.0,0.5,
    0.5,1.0,-0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkf[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkf[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkf[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotYStart failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotZCenter(surf, theta);
float checkg[12] = {
    1.0,0.0,-0.5,
    1.0,1.0,-0.5,
    1.0,0.0,0.5,
    0.0,0.0,-0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkg[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkg[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkg[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotZCenter failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotZOrigin(surf, theta);
float checkh[12] = {
    0.0,1.0,-0.5,
    -1.0,1.0,-0.5,
    0.0,1.0,0.5,
    0.0,0.0,-0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {

```

```

    if (ISEQUALF(checkh[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkh[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkh[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotZOrigin failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotZStart(surf, theta);
float checki[12] = {
    0.0,1.0,-0.5,
    0.0,0.0,-0.5,
    0.0,1.0,0.5,
    1.0,1.0,-0.5
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checki[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checki[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checki[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotZStart failed");
        PBErrCatch(BCurveErr);
    }
}
VecFloat3D axis = VecFloatCreateStatic3D();
VecSet(&axis, 0, 1.0); VecSet(&axis, 1, 1.0); VecSet(&axis, 2, 1.0);
VecNormalise(&axis);
BBodyRotAxisCenter(surf, &axis, theta);
float checkj[12] = {
    -0.122009,0.666667,-0.044658,
    0.122008,0.333334,-0.955342,
    0.788675,0.422650,0.288675,
    0.211325,1.577350,-0.288675
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkj[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||
        ISEQUALF(checkj[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
        false ||
        ISEQUALF(checkj[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
        false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "BBodyRotAxisCenter failed");
        PBErrCatch(BCurveErr);
    }
}
BBodyRotAxisOrigin(surf, &axis, theta);
float checkk[12] = {
    -0.244017,0.122008,0.622008,
    -0.910684,0.455342,-0.044658,
    0.422650,0.788675,0.288675,
    -0.577350,0.788675,1.288675
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
    if (ISEQUALF(checkk[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
        false ||

```

```

ISEQUALF(checkk[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
false ||
ISEQUALF(checkk[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
false) {
BCurveErr->_type = PBErrTypeUnitTestFailed;
sprintf(BCurveErr->_msg, "BBodyRotAxisOrigin failed");
PBErrCatch(BCurveErr);
}
}
BBodyRotAxisStart(surf, &axis, theta);
float checkl[12] = {
-0.244017, 0.122008, 0.622008,
-1.154700, -0.211325, 0.866026,
-0.488034, 1.032692, 0.955342,
0.089317, -0.122008, 1.532692
};
for (int iCtrl = BBodyGetNbCtrl(surf); iCtrl--;) {
if (ISEQUALF(checkl[3 * iCtrl], surf->_ctrl[iCtrl]->_val[0]) ==
false ||
ISEQUALF(checkl[3 * iCtrl + 1], surf->_ctrl[iCtrl]->_val[1]) ==
false ||
ISEQUALF(checkl[3 * iCtrl + 2], surf->_ctrl[iCtrl]->_val[2]) ==
false) {
BCurveErr->_type = PBErrTypeUnitTestFailed;
sprintf(BCurveErr->_msg, "BBodyRotAxisStart failed");
PBErrCatch(BCurveErr);
}
}
BBodyFree(&surf);
printf("UnitTestBBodyRotate OK\n");
}

VecFloat* PointCloud(VecFloat* input) {
VecFloat* v = VecFloatCreate(3);
float x = VecGet(input, 0);
float y = VecGet(input, 1);
VecSet(v, 0, x);
VecSet(v, 1, y);
VecSet(v, 2, x * x + y * y);
return v;
}

void UnitTestBBodyFromPointCloud() {
GSetVecFloat inputs = GSetVecFloatCreateStatic();
GSetVecFloat outputs = GSetVecFloatCreateStatic();
int nbPts = 20;
int dimIn = 2;
for (int iPt = 0; iPt < nbPts; ++iPt) {
VecFloat* input = VecFloatCreate(dimIn);
VecSet(input, 0, rnd());
VecSet(input, 1, rnd());
VecFloat* output = PointCloud(input);
VecSet(output, 2, VecGet(output, 2) + 0.1 * (rnd() - 0.5));
GSetAppend(&inputs, input);
GSetAppend(&outputs, output);
}
float bias;
int order = 1;
BBody* bbody = BBodyFromPointCloud(order, &inputs, &outputs, &bias);
BBodyPrint(bbody, stdout); printf("\n");
printf("bias %f\n", bias);
BBodyFree(&bbody);
}

```

```

order = 2;
bbody = BBodyFromPointCloud(order, &inputs, &outputs, &bias);
BBodyPrint(bbody, stdout);printf("\n");
printf("bias %f\n", bias);
BBodyFree(&bbody);
order = 3;
bbody = BBodyFromPointCloud(order, &inputs, &outputs, &bias);
BBodyPrint(bbody, stdout);printf("\n");
printf("bias %f\n", bias);
for (int iPt = 0; iPt < nbPts; ++iPt) {
    VecFloat* input = VecFloatCreate(dimIn);
    VecSet(input, 0, rnd());
    VecSet(input, 1, rnd());
    VecFloat* output = PointCloud(input);
    VecFloat* prev = BBodyGet(bbody, input);
    bias = VecDist(output, prev);
    VecPrint(input, stdout);
    printf(" -> ");
    VecPrint(output, stdout);
    printf(",");
    VecPrint(prev, stdout);
    printf(" %f\n", bias);
    VecFree(&input);
    VecFree(&output);
    VecFree(&prev);
}
BBodyFree(&bbody);
while (GSetNbElem(&inputs) > 0) {
    VecFloat* v = GSetPop(&inputs);
    VecFree(&v);
}
while (GSetNbElem(&outputs) > 0) {
    VecFloat* v = GSetPop(&outputs);
    VecFree(&v);
}
printf("UnitTestBBodyFromPointCloud OK\n");
}

void UnitTestBBody() {
    UnitTestBBodyCreateFree();
    UnitTestBBodyGetSet();
    UnitTestBBodyGet();
    UnitTestBBodyClone();
    UnitTestBBodyPrint();
    UnitTestBBodyLoadSave();
    UnitTestBBodyGetCenter();
    UnitTestBBodyTranslate();
    UnitTestBBodyScale();
    UnitTestBBodyGetBoundingBox();
    UnitTestBBodyRotate();
    UnitTestBBodyFromPointCloud();
    printf("UnitTestBBody OK\n");
}

void UnitTestAll() {
    UnitTestBCurve();
    UnitTestSCurve();
    UnitTestSCurveIter();
    UnitTestBBody();
    printf("UnitTestAll OK\n");
}

```

```

int main() {
    UnitTestAll();

    // Return success code
    return 0;
}

```

## 6 Unit tests output

```

UnitTestBCurveCreateCloneFree OK
order(3) dim(2) <0.000,1.000> <2.000,3.000> <4.000,5.000> <6.000,7.000>
UnitTestBCurveLoadSavePrint OK
UnitTestBCurveGetSetCtrl OK
UnitTestBCurveGet OK
UnitTestBCurveGetOrderDimNbCtrl OK
UnitTestBCurveGetApproxLenCenter OK
UnitTestBCurveRot OK
UnitTestBCurveScale OK
UnitTestBCurveTranslate OK
UnitTestBCurveFromCloudPoint OK
UnitTestBCurveGetWeightCtrlPt OK
UnitTestBCurveGetBoundingBox OK
UnitTestBCurve OK
UnitTestSCurveCreateCloneFree OK
order(3) dim(2) nbSeg(3) <<0.000000,1.000000>> <<2.000000,3.000000>> <<4.000000,5.000000>> <<6.000000,7.000000>> <<8.000000,9.000000>>
UnitTestSCurveLoadSavePrint OK
UnitTestSCurveGetSetCtrl OK
UnitTestSCurveGetAddRemoveSeg OK
UnitTestSCurveGet OK
UnitTestSCurveGetOrderDimNbSegMaxUNbCtrl OK
UnitTestSCurveGetApproxLenCenter OK
UnitTestSCurveRot OK
UnitTestSCurveScale OK
UnitTestSCurveTranslate OK
UnitTestSCurveGetBoundingBox OK
UnitTestSCurveGetNewDim OK
order(1) dim(2) nbSeg(4) <<1.000000,2.000000>> <<4.000000,6.000000>> <<-1.000000,12.000000>> <<-4.000000,8.000000>> <<4.000000,12.000000>>
Type: Facoid
Dim: 2
Pos: <1.000,2.000>
Axis(0): <3.000,4.000>
Axis(1): <-5.000,6.000>
UnitTestSCurveCreateFromShapoid OK
UnitTestSCurveGetDDistToCurve OK
Curve before Chaikin: order(1) dim(2) nbSeg(2) <<0.000000,1.000000>> <<5.000000,4.000000>> <<2.000000,3.000000>>
Curve after Chaikin: order(1) dim(2) nbSeg(5) <<0.000000,1.000000>> <<4.050000,3.430000>> <<4.520000,3.720000>> <<4.000000,1.000000>>
UnitTestSCurveChaikin OK
UnitTestSCurve OK
UnitTestSCurveIterCreate OK
UnitTestSCurveIterSetGet OK
UnitTestSCurveStep OK
UnitTestSCurveIter OK
UnitTestBBodyCreateFree OK
UnitTestBBodyGetSet OK
UnitTestBBodyGet OK
UnitTestBBodyClone OK
order(1) dim(<2,3>) <0.000,0.000,0.000> <0.000,1.000,0.000><1.000,0.000,0.000><0.000,0.000,1.000>
UnitTestBBodyPrint OK

```



```

UnitTestBBodyLoadSave OK
UnitTestBBodyGetCenter OK
UnitTestBBodyTranslate OK
UnitTestBBodyScale OK
UnitTestBBodyGetBoundingBox OK
UnitTestBBodyRotate OK
order(1) dim(<2,3>) <-0.000,0.000,-0.381> <0.000,1.000,0.760><1.000,0.000,0.590><1.000,1.000,1.621>
bias 0.133216
order(2) dim(<2,3>) <0.000,0.000,0.099> <-0.000,0.500,-0.183> <-0.000,1.000,0.993><0.500,0.000,0.015><0.500,0.500,0.015>
bias 0.027518
order(3) dim(<2,3>) <-0.000,-0.000,0.033> <0.000,0.334,0.442> <-0.000,0.666,-0.432> <0.000,1.000,1.241><0.334,0.002,0.442>
bias 0.022200
<0.238,0.971> -> <0.238,0.971,0.999>,<0.238,0.971,0.884> 0.115122
<0.902,0.851> -> <0.902,0.851,1.538>,<0.902,0.851,1.426> 0.111708
<0.267,0.540> -> <0.267,0.540,0.362>,<0.267,0.540,0.362> 0.000136
<0.375,0.760> -> <0.375,0.760,0.719>,<0.375,0.760,0.736> 0.017135
<0.513,0.668> -> <0.513,0.668,0.709>,<0.513,0.668,0.725> 0.016894
<0.532,0.039> -> <0.532,0.039,0.284>,<0.532,0.039,0.675> 0.391306
<0.438,0.932> -> <0.438,0.932,1.060>,<0.438,0.932,1.032> 0.028135
<0.931,0.721> -> <0.931,0.721,1.386>,<0.931,0.721,1.358> 0.028535
<0.284,0.739> -> <0.284,0.739,0.626>,<0.284,0.739,0.631> 0.005236
<0.640,0.354> -> <0.640,0.354,0.535>,<0.640,0.354,0.616> 0.080872
<0.688,0.166> -> <0.688,0.166,0.501>,<0.688,0.166,0.654> 0.153337
<0.440,0.880> -> <0.440,0.880,0.968>,<0.440,0.880,0.964> 0.004035
<0.829,0.330> -> <0.829,0.330,0.797>,<0.829,0.330,0.804> 0.006970
<0.229,0.893> -> <0.229,0.893,0.851>,<0.229,0.893,0.788> 0.062305
<0.350,0.687> -> <0.350,0.687,0.594>,<0.350,0.687,0.615> 0.020651
<0.956,0.589> -> <0.956,0.589,1.261>,<0.957,0.589,1.317> 0.055367
<0.657,0.859> -> <0.657,0.859,1.169>,<0.657,0.859,1.182> 0.012453
<0.440,0.924> -> <0.440,0.924,1.047>,<0.440,0.924,1.024> 0.023220
<0.398,0.815> -> <0.398,0.815,0.823>,<0.398,0.815,0.831> 0.008730
<0.684,0.911> -> <0.684,0.911,1.298>,<0.684,0.911,1.319> 0.021361
UnitTestBBodyFromPointCloud OK
UnitTestBBody OK
UnitTestAll OK

```

bcurve.txt:

```

{
  "_order": "3",
  "_dim": "2",
  "_ctrl": [
    {
      "_dim": "2",
      "_val": ["0.000000", "1.000000"]
    },
    {
      "_dim": "2",
      "_val": ["2.000000", "3.000000"]
    },
    {
      "_dim": "2",
      "_val": ["4.000000", "5.000000"]
    },
    {
      "_dim": "2",
      "_val": ["6.000000", "7.000000"]
    }
  ]
}

```

scurve.txt:

```
{
  "_order": "3",
  "_dim": "2",
  "_nbSeg": "3",
  "_ctrl": [
    {
      "_dim": "2",
      "_val": ["0.000000", "1.000000"]
    },
    {
      "_dim": "2",
      "_val": ["2.000000", "3.000000"]
    },
    {
      "_dim": "2",
      "_val": ["4.000000", "5.000000"]
    },
    {
      "_dim": "2",
      "_val": ["6.000000", "7.000000"]
    },
    {
      "_dim": "2",
      "_val": ["8.000000", "9.000000"]
    },
    {
      "_dim": "2",
      "_val": ["10.000000", "11.000000"]
    },
    {
      "_dim": "2",
      "_val": ["12.000000", "13.000000"]
    },
    {
      "_dim": "2",
      "_val": ["14.000000", "15.000000"]
    },
    {
      "_dim": "2",
      "_val": ["16.000000", "17.000000"]
    },
    {
      "_dim": "2",
      "_val": ["18.000000", "19.000000"]
    }
  ]
}
```

bbody.txt:

```
{
  "_order": "1",
  "_dim": {
    "_dim": "2",
    "_val": ["2", "3"]
  }
}
```

```

},
"_ctrl":[
  {
    "_dim":"3",
    "_val":["0.000000","0.000000","0.000000"]
  },
  {
    "_dim":"3",
    "_val":["0.000000","1.000000","0.000000"]
  },
  {
    "_dim":"3",
    "_val":["1.000000","0.000000","0.000000"]
  },
  {
    "_dim":"3",
    "_val":["0.000000","0.000000","1.000000"]
  }
]
}

```