

BCurve

P. Baillehache

February 3, 2018

Contents

1	Definitions	2
1.1	BCurve definition	2
1.2	BCurve from cloud points	2
2	Interface	3
3	Code	10
3.1	bcurve.c	10
3.2	bcurve-inline.c	24
4	Makefile	39
5	Unit tests	40
6	Unit tests output	57

Introduction

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

It offers function to create, clone, load, save and modify a curve, to print it, to scale, rotate (in 2D) or translate it, to get its approximate length (sum of distance between control points), to create a BCurve connecting points of a point cloud, to get the weights (coefficients of each control point given the value of the parameter of the curve), and to get the bounding box.

The library also includes a SCurve structure which is a set of BCurve (called segments) continuously connected and has the same interface as for

a BCurve, plus function to add and remove segments.

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

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

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 $\overrightarrow{C_{N-1}} = \overrightarrow{P_{N-1}}$, and others \vec{C}_i can be obtained by solving the linear system below for each dimension:

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

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 =====

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

#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 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)

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

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

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

/*typedef struct BSurf {
    // Order
    int _order;
    // Dimensions (input/output)
    VecShort *_dim;
    // ((_order + 1) ^ _dim[0]) control points of the surface
    VecFloat **_ctrl;
} BSurf;*/

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

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

// Clone the BCurve

```

```

BCurve* BCurveClone(BCurve *that);

// 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 *stream);

// Save the BCurve to the stream
// Return true upon success, false else
bool BCurveSave(BCurve *that, FILE *stream);

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

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

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

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

// Get the iCtrl-th control point
#if BUILDMODE != 0
inline
#endif
VecFloat* BCurveCtrl(BCurve *that, int iCtrl);

// Get the value of the BCurve at parameter 'u' (in [0.0, 1.0])
VecFloat* BCurveGet(BCurve *that, float u);

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

```

```

void BCurveScaleCenterScalar(BCurve *that, float c);

// Translate the curve by 'v'
#if BUILDMODE != 0
inline
#endif
void BCurveTranslate(BCurve *that, VecFloat *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(GSet *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(BCurve *that, float t);

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

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

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

// 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 *stream);

// Save the SCurve to the stream
// Return true upon success, false else
bool SCurveSave(SCurve *that, FILE *stream);

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

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

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

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

// Get the order of the SCurve

```

```

#if BUILDMODE != 0
inline
#endif
int SCurveGetOrder(SCurve *that);

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

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

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

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

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

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

// 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 *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 *that);

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

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

// Get the approximate length of the SCurve (sum of approxLen
// of its BCurves)
#if BUILDMODE != 0

```



```

inline
#endif
float SCurveGetApproxLen(SCurve *that);

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

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

// Return the max value for the parameter 'u' of SCurveGet
#if BUILDMODE != 0
inline
#endif
float SCurveGetMaxU(SCurve *that);

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

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

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

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

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

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

```

```

void SCurveScaleOriginScalar(SCurve *that, float c);

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

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

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

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

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

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

#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

```

```

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

// Create a new BCurve of order 'order' and dimension 'dim'
BCurve* BCurveCreate(int order, int dim) {
#ifdef 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
    that->_dim = dim;
    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(BCurve *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
    clone->_dim = that->_dim;
    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;
}

// 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 *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
// If 'that' is already allocated
if (*that != NULL)
    // Free memory
    BCurveFree(that);
// Read the order and dimension
int order;
int dim;
int ret = fscanf(stream, "%d %d", &order, &dim);
// If we couldn't read
if (ret == EOF)
    return false;
// Allocate memory
*that = BCurveCreate(order, dim);
// For each control point
for (int iCtrl = 0; iCtrl < (order + 1); ++iCtrl) {
    // Load the control point
    ret = VecLoad((*that)->_ctrl + iCtrl, stream);
    // If we couldn't read the control point or the control point
    // is not of the correct dimension
    if (ret == false || VecDim((*that)->_ctrl[iCtrl]) != (*that)->_dim)
        return false;
}
// Return success code
return true;
}

// Save the BCurve to the stream
// Return true upon success, false else
bool BCurveSave(BCurve *that, FILE *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
    // Save the order and dimension
    int ret = fprintf(stream, "%d %d\n", that->_order, that->_dim);
    // If the fprintf failed
    if (ret < 0)
        // Stop here
        return false;
    // For each control point
    for (int iCtrl = 0; iCtrl < that->_order + 1; ++iCtrl) {
        // Save the control point
        ret = VecSave(that->_ctrl[iCtrl], stream);
        // If we couldn't save the control point

```

```

        if (ret == false)
            // Stop here
            return false;
    }
    // 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(BCurve *that, FILE *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' (in [0.0, 1.0])
VecFloat* BCurveGet(BCurve *that, float u) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (u < 0.0 - PBMath_EPSILON || u > 1.0 + PBMath_EPSILON) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'u' is invalid (0.0<=%f<=1.0)", u);
        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(GSet *set) {
    #if BUILDMODE == 0
        if (set == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'set' is null");
            PBErrCatch(BCurveErr);
        }
        if (set->_nbElem < 1) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'set' is empty");
            PBErrCatch(BCurveErr);
        }
    }
    #endif
    // Declare a variable to memorize the result
    int order = set->_nbElem - 1;
    int dim = VecDim((VecFloat*)(set->_head->_data));
    BCurve *curve = BCurveCreate(order, dim);
    // Set the first control point to the first point in the point cloud
    BCurveSetCtrl(curve, 0, (VecFloat*)(set->_head->_data));
    // 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, (VecFloat*)(set->_tail->_data));
        // 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(set->nbElem);
// Set the dimensions of the matrix of the linear system
VecSet(&dimMat, 0, order - 1);
VecSet(&dimMat, 1, order - 1);
// For each point
GSetElem *elem = set->_head->_next;
int iPoint = 1;
while (elem != NULL) {
    // Get the distance from the previous point
    float d = VecDist((VecFloat*)(elem->_prev->_data),
        (VecFloat*)(elem->_data));
    VecSet(t, iPoint, d + VecGet(t, iPoint - 1));
    ++iPoint;
    elem = elem->_next;
}
// Normalize t
for (iPoint = 1; iPoint <= order; ++iPoint)
    VecSet(t, iPoint, VecGet(t, iPoint) / VecGet(t, order));
// For each dimension
for (int 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;
        VecSet(&dimMat, 1, VecGet(&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;
            VecSet(&dimMat, 0, VecGet(&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((VecFloat*)(set->_head->_data), iDim);
        x -= VecGet(weight, order) *
            VecGet((VecFloat*)(set->_tail->_data), 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(BCurve *that, float t) {
    #if 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(BCurve *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(that->_dim);
    // For each dimension
    for (int iDim = that->_dim; iDim--;) {
        // For each control point
        for (int iCtrl = that->_order + 1; iCtrl--;) {
            // If it's the first control point in this dimension
            if (iCtrl == that->_order) {
                // Initialise the bounding box
                VecSet(res->_s._pos, iDim, VecGet(that->_ctrl[iCtrl], iDim));
                VecSet(res->_s._axis[iDim], iDim,
                    VecGet(that->_ctrl[iCtrl], iDim));
            }
            // Else, it's not the first control point in this dimension
        } else {
            // 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) >
                VecGet(ShapoidAxis(res, iDim), iDim))
                VecSet(ShapoidAxis(res, iDim), iDim,
                    VecGet(that->_ctrl[iCtrl], iDim));
        }
    }
    VecSet(ShapoidAxis(res, iDim), iDim,
        VecGet(ShapoidAxis(res, iDim), iDim) -
        VecGet(ShapoidPos(res), iDim));
}
// Return the result
return res;
}

// Create a new SCurve of dimension 'dim', order 'order' and
// 'nbSeg' segments
SCurve* SCurveCreate(int order, int dim, int nbSeg) {
    #if BUILDMODE == 0
        if (order < 0) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "Invalid order (%d>=0)", order);
            PBErrCatch(BCurveErr);
        }
    #endif
}

```

```

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
that->_dim = dim;
that->_order = order;
that->_nbSeg = nbSeg;
// Create the GSet
that->_ctrl = GSetCreateStatic();
that->_seg = GSetCreateStatic();
// 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] = (VecFloat*)(that->_ctrl._head->_data);
        // Add the control points
        for (int iCtrl = order; iCtrl--;)
            GSetPush(&(that->_ctrl), BCurveCtrl(seg, iCtrl));
    }
    // Else, it's the first segment
    else {
        // Add the control points
        for (int iCtrl = order + 1; iCtrl--;)
            GSetPush(&(that->_ctrl), BCurveCtrl(seg, iCtrl));
    }
    // Add the segment
    GSetPush(&(that->_seg), seg);
}
// Return the new SCurve
return that;
}

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

```

```

        VecFloat *ctrlClone = (VecFloat*)GSetIterGet(&iterClone);
        VecCopy(ctrlClone, ctrl);
    } while (GSetIterStep(&iter) && GSetIterStep(&iterClone));
    return clone;
}

// 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 *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
    // If 'that' is already allocated
    if (*that != NULL)
        // Free memory
        SCurveFree(that);
    // Read the number of segment, order and dimension
    int nbSeg;
    int order;
    int dim;
    int ret = fscanf(stream, "%d %d %d", &order, &dim, &nbSeg);
    // If we couldn't read
    if (ret == EOF)
        return false;
    // If data are invalid
    if (nbSeg < 1 || order < 0 || dim < 1)
        return false;
    // Allocate memory
    *that = SCurveCreate(order, dim, nbSeg);
    // For each control point
    GSetIterForward iter = GSetIterForwardCreateStatic(&((*that)->_ctrl));
    do {
        // Load the control point
        VecFloat *loadCtrl = NULL;
        ret = VecLoad(&loadCtrl, stream);
        // If we couldn't read the control point or the control point
        // is not of the correct dimension
        if (ret == false || VecDim(loadCtrl) != (*that)->_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);
    } while (GSetIterStep(&iter));
    // Return success code
    return true;
}

// Save the SCurve to the stream
// Return true upon success, false else
bool SCurveSave(SCurve *that, FILE *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
// Save the nb of segment, order and dimension
int ret = fprintf(stream, "%d %d %d\n",
    that->_order, that->_dim, that->_nbSeg);
// If the fprintf failed
if (ret < 0)
    // Stop here
    return false;
// For each control point
GSetIterForward iter = GSetIterForwardCreateStatic(&(that->_ctrl));
do {
    VecFloat *ctrl = (VecFloat*)GSetIterGet(&iter);
    // Save the control point
    ret = VecSave(ctrl, stream);
    // If we couldn't save the control point
    if (ret == false)
        // Stop here
        return false;
} while (GSetIterStep(&iter));
// 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(SCurve *that, FILE *stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 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);
        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 *that) {
#ifdef 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] = that->_ctrl._tail->_data;
    // 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 *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] = that->_ctrl._head->_data;
    // 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 *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 *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(BCurveErr->_msg, "'that' is null");
        PBErCatch(BCurveErr);
    }
    if (that->_nbSeg < 2) {
        BCurveErr->_type = PBErTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'that' has only one segment");
        PBErCatch(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.
Facoid* SCurveGetBoundingBox(SCurve *that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PBErCatch(BCurveErr);
        }
    #endif
    // Declare a set to memorize the bounding box of each segment
    GSet set = GSetCreateStatic();
    // 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;
}

```

3.2 bcurve-inline.c

```
// ===== BCURVE-INLINE.C =====

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

// Set the value of the iCtrl-th control point to v
#if BUILDMODE != 0
inline
#endif
void BCurveSetCtrl(BCurve *that, int iCtrl, VecFloat *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 (VecDim(v) != BCurveGetDim(that)) {
            BCurveErr->_type = PBErrTypeInvalidArg;
            sprintf(BCurveErr->_msg, "'v' 's dimension is invalid (%d<%d)",
                    VecDim(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
inline
#endif
VecFloat* BCurveGetCtrl(BCurve *that, int iCtrl) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PBErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' 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);
        }
    #endif
    // Return a copy of the control point
    return VecClone(that->_ctrl[iCtrl]);
}

// Get the iCtrl-th control point
```



```

#if BUILDMODE != 0
inline
#endif
VecFloat* BCurveCtrl(BCurve *that, int iCtrl) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' 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);
    }
#endif
    // Return the control point
    return that->_ctrl[iCtrl];
}

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

```

```

    }
#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
inline
#endif
VecFloat* BCurveGetCenter(BCurve *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
inline
#endif
void BCurveRotOrigin(BCurve *that, 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
inline

```

```

#endif
void BCurveRotStart(BCurve *that, float theta) {
#ifdef 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
#ifdef BUILDMODE != 0
inline
#endif
void BCurveRotCenter(BCurve *that, float theta) {
#ifdef 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
#ifdef BUILDMODE != 0

```

```

inline
#endif
void BCurveScaleOriginVector(BCurve *that, VecFloat *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);
    }
#endif
    // For each control point
    for (int iCtrl = that->_order + 1; iCtrl--;) {
        VecFloat *ctrl = that->_ctrl[iCtrl];
        // Scale the control point
        for (int dim = 0; dim < VecDim(ctrl); ++dim)
            VecSet(ctrl, dim, VecGet(ctrl, dim) * VecGet(v, dim));
    }
}

// Scale the curve by 'c' relatively to the origin
#if BUILDMODE != 0
inline
#endif
void BCurveScaleOriginScalar(BCurve *that, 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
inline
#endif
void BCurveScaleStartVector(BCurve *that, VecFloat *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);
    }
#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 (int dim = 0; dim < VecDim(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
inline
#endif
void BCurveScaleStartScalar(BCurve *that, float c) {
    #if BUILDMODE == 0
        if (that == NULL) {
            BCurveErr->_type = PErrTypeNullPointer;
            sprintf(BCurveErr->_msg, "'that' is null");
            PErrCatch(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
inline
#endif
void BCurveScaleCenterVector(BCurve *that, VecFloat *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);
        }
    #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 (int dim = 0; dim < VecDim(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
inline
#endif
void BCurveScaleCenterScalar(BCurve *that, float c) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        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
        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
inline
#endif
void BCurveTranslate(BCurve *that, VecFloat *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);
    }
#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);
}

// Get the number of BCurve in the SCurve
#if BUILDMODE != 0
inline

```

```

#endif
int SCurveGetNbSeg(SCurve *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
inline
#endif
int SCurveGetDim(SCurve *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
inline
#endif
int SCurveGetOrder(SCurve *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
inline
#endif
VecFloat* SCurveGetCtrl(SCurve *that, 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((VecFloat*)GSetGet(&(that->_ctrl), iCtrl));
}

```

```

}

// Get the 'iCtrl'-th control point
#if BUILDMODE != 0
inline
#endif
VecFloat* SCurveCtrl(SCurve *that, 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 (VecFloat*)GSetGet(&(that->_ctrl), iCtrl);
}

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

// Get the 'iSeg'-th segment
#if BUILDMODE != 0
inline
#endif
BCurve* SCurveSeg(SCurve *that, int iSeg) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (iSeg < 0 || iSeg >= that->_nbSeg) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iSeg' is invalid (0<=%d<%d)",
            iSeg, that->_nbSeg);
        PBErrCatch(BCurveErr);
    }
#endif
}

```



```

    return (BCurve*)GSetGet(&(that->_seg), iSeg);
}

// Return the center of the SCurve (average of control points)
#if BUILDMODE != 0
inline
#endif
VecFloat* SCurveGetCenter(SCurve *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 = 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
#if BUILDMODE != 0
inline
#endif
float SCurveGetMaxU(SCurve *that) {
#if 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
#if BUILDMODE != 0
inline
#endif
int SCurveGetNbCtrl(SCurve *that) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#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
inline
#endif

```

```

void SCurveRotOrigin(SCurve *that, float theta) {
#ifdef 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
#ifdef BUILDMODE != 0
inline
#endif
void SCurveRotStart(SCurve *that, float theta) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat *origin = (VecFloat*)(that->_ctrl._head->_data);
    // 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
#ifdef BUILDMODE != 0
inline
#endif
void SCurveRotCenter(SCurve *that, float theta) {
#ifdef 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(&(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
inline
#endif
void SCurveScaleOriginVector(SCurve *that, VecFloat *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(&(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
inline
#endif
void SCurveScaleOriginScalar(SCurve *that, 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(&(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
inline
#endif

```

```

void SCurveScaleStartVector(SCurve *that, VecFloat *v) {
#ifdef 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 (VecDim(v) != SCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' 's dimension is invalid (%d=%d)",
            VecDim(v), SCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat *origin = (VecFloat*)(that->_ctrl._head->_data);
    // 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)
#ifdef BUILDMODE != 0
inline
#endif
void SCurveScaleStartScalar(SCurve *that, float c) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
#endif
    VecFloat *origin = (VecFloat*)(that->_ctrl._head->_data);
    // 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
            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
inline
#endif
void SCurveScaleCenterVector(SCurve *that, VecFloat *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 (VecDim(v) != SCurveGetDim(that)) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'v' 's dimension is invalid (%d=%d)",
            VecDim(v), SCurveGetDim(that));
        PBErrCatch(BCurveErr);
    }
#endif
    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);
        // 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
inline
#endif
void SCurveScaleCenterScalar(SCurve *that, 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(&(amp;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
inline
#endif
void SCurveTranslate(SCurve *that, VecFloat *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);
    }
#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' (in [0.0, _nbSeg])
// The value is equal to the value of the floor(u)-th segment at
// value (u - floor(u))
#if BUILDMODE != 0
inline
#endif
VecFloat* SCurveGet(SCurve *that, float u) {
#if BUILDMODE == 0
    if (that == NULL) {
        BCurveErr->_type = PBErrTypeNullPointer;
        sprintf(BCurveErr->_msg, "'that' is null");
        PBErrCatch(BCurveErr);
    }
    if (u < 0.0 - PBMath_EPSILON ||
        u > (float)(that->_nbSeg) + PBMath_EPSILON) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'u' is invalid (0.0<=%f<=%d.0)",
            u, that->_nbSeg);
        PBErrCatch(BCurveErr);
    }
#endif
    // Get the segment the corresponding to 'u'
    int iSeg = (int)floor(u);
    // Get the value of 'u' in this segment
    u -= (float)iSeg;
    // Get the value of the BCurve
    return BCurveGet(SCurveSeg(that, iSeg), u);
}

```

```

}

// Get the approximate length of the SCurve (sum of approxLen
// of its BCurves)
#if BUILDMODE != 0
inline
#endif
float SCurveGetApproxLen(SCurve *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
inline
#endif
void SCurveSetCtrl(SCurve *that, int iCtrl, VecFloat *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 >= SCurveGetNbCtrl(that)) {
        BCurveErr->_type = PBErrTypeInvalidArg;
        sprintf(BCurveErr->_msg, "'iCtrl' is invalid (0<=%d<=%d)",
            iCtrl, SCurveGetNbCtrl(that));
        PBErrCatch(BCurveErr);
    }
#endif
    VecCopy((VecFloat*)GSetGet(&(that->_ctrl), iCtrl), v);
}

```

4 Makefile

```

#directory
PBERDIR=../PBErr

# Build mode

```

```

# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILDMODE=1

include $(PBERRDIR)/Makefile.inc

INCPATH=-I./ -I$(PBERRDIR)/
BUILDOPTIONS=$(BUILDPARAM) $(INCPATH)

# compiler
COMPILER=gcc

#rules
all : main

main: main.o pberr.o pbmath.o Makefile
$(COMPILER) main.o pberr.o pbmath.o $(LINKOPTIONS) -o main

main.o : main.c $(PBERRDIR)/pberr.h pbmath.h pbmath-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c main.c

pbmath.o : pbmath.c pbmath.h pbmath-inline.c Makefile
$(COMPILER) $(BUILDOPTIONS) -c pbmath.c

pberr.o : $(PBERRDIR)/pberr.c $(PBERRDIR)/pberr.h Makefile
$(COMPILER) $(BUILDOPTIONS) -c $(PBERRDIR)/pberr.c

clean :
rm -rf *.o main

valgrind :
valgrind -v --track-origins=yes --leak-check=full --gen-suppressions=yes --show-leak-kinds=all ./main

unitTest :
main > unitTest.txt; diff unitTest.txt unitTestRef.txt

```

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) {
        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 UnitTestBCurveGetOrderDim() {
    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);
    }
    BCurveFree(&curve);
    printf("UnitTestBCurveGetOrderDim 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(VecGet(BCurveCtrl(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(VecGet(BCurveCtrl(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(VecGet(BCurveCtrl(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(VecGet(BCurveCtrl(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(VecGet(BCurveCtrl(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(VecGet(BCurveCtrl(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(VecGet(BCurveCtrl(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);
    GSet *set = GSetCreate();
    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");
        PBErrCatch(BCurveErr);
    }
    ShapoidFree(&bound);
    ShapoidFree(&check);
    BCurveFree(&curve);
    VecFree(&v);
    printf("UnitTestBCurveGetBoundingBox OK\n");
}

void UnitTestBCurve() {
    UnitTestBCurveCreateCloneFree();
    UnitTestBCurveLoadSavePrint();
    UnitTestBCurveGetSetCtrl();
    UnitTestBCurveGet();
    UnitTestBCurveGetOrderDim();
    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 = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveCreate failed");
        PBErrCatch(BCurveErr);
    }
    VecFloat *v = VecFloatCreate(dim);
    GSetIterForward iter = GSetIterForwardCreateStatic(&(curve->_ctrl));
    do {
        VecFloat *ctrl = GSetIterGet(&iter);
        if (VecIsEqual(ctrl, v) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveCreate failed");
            PBErrCatch(BCurveErr);
        }
    } while (GSetIterStep(&iter));
    iter = GSetIterForwardCreateStatic(&(curve->_seg));
    VecFloat *prevCtrl = (VecFloat*)(curve->_ctrl._head->_data);
    do {
        BCurve *seg = GSetIterGet(&iter);
        if (seg->_ctrl[0] != prevCtrl) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveCreate failed");
            PBErrCatch(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) {
        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");
        PBErrCatch(BCurveErr);
    }
    fclose(file);
    if (load->_dim != dim || load->_order != order ||
        load->_order != order) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveLoad failed");
        PBErrCatch(BCurveErr);
    }
    iter = GSetIterForwardCreateStatic(&(curve->_ctrl));
    GSetIterForward iterLoad =
        GSetIterForwardCreateStatic(&(load->_ctrl));
    do {
        VecFloat *ctrl = GSetIterGet(&iter);
        VecFloat *ctrlLoad = GSetIterGet(&iterLoad);
        if (VecIsEqual(ctrl, ctrlLoad) == false) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;

```

```

        sprintf(BCurveErr->_msg, "SCurveLoad failed");
        PBErrCatch(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);
    for (int iCtrl = SCurveGetNbCtrl(curve); iCtrl--;) {
        for (int iDim = dim; iDim--;)
            VecSet(v, iDim, iCtrl * dim + iDim);
        SCurveSetCtrl(curve, iCtrl, v);
    }
    GSetIterForward iter = GSetIterForwardCreateStatic(&(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 = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveSetCtrl failed");
            PBErrCatch(BCurveErr);
        }
        if (ctrl != SCurveCtrl(curve, iCtrl)) {
            BCurveErr->_type = PBErrTypeUnitTestFailed;
            sprintf(BCurveErr->_msg, "SCurveCtrl failed");
            PBErrCatch(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--;)
            VecSet(SCurveCtrl(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--;)
            VecSet(SCurveCtrl(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--;)
            VecSet(SCurveCtrl(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--;)
        VecSet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0), 0),
        pa[iCtrl * 2]) == false ||
        ISEQUALF(VecGet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0), 0),
        pb[iCtrl * 2]) == false ||
        ISEQUALF(VecGet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0), 0),
        pc[iCtrl * 2]) == false ||
        ISEQUALF(VecGet(SCurveCtrl(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--;)
            VecSet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0),
    pa[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0),
    pb[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0),
    pc[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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--;)
    VecSet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0),
    pd[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0), 0),
    pe[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0), 0),
    pf[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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--;)
      VecSet(SCurveCtrl(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(VecGet(SCurveCtrl(curve, iCtrl), 0),
    p[iCtrl * 2]) == false ||
    ISEQUALF(VecGet(SCurveCtrl(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--;) {
        VecSet(SCurveCtrl(curve, iCtrl), 0,
            cos(PBMATH_QUARTERPI * (float)iCtrl * 0.5));
        VecSet(SCurveCtrl(curve, iCtrl), 1,
            sin(PBMATH_QUARTERPI * (float)iCtrl * 0.5));
    }
    Facoid *bound = SCurveGetBoundingBox(curve);
    if (ISEQUALF(VecGet(ShapoidPos(bound), 0), -1.0) == false ||
        ISEQUALF(VecGet(ShapoidPos(bound), 1), -0.382683) == false ||
        ISEQUALF(VecGet(ShapoidAxis(bound, 0), 0), 2.0) == false ||
        ISEQUALF(VecGet(ShapoidAxis(bound, 0), 1), 0.0) == false ||
        ISEQUALF(VecGet(ShapoidAxis(bound, 1), 0), 0.0) == false ||
        ISEQUALF(VecGet(ShapoidAxis(bound, 1), 1), 1.382683) == false) {
        BCurveErr->_type = PBErrTypeUnitTestFailed;
        sprintf(BCurveErr->_msg, "SCurveGetBoundingBox failed");
        PBErrCatch(BCurveErr);
    }
    ShapoidFree(&bound);
    SCurveFree(&curve);
    printf("UnitTestSCurveGetBoundingBox OK\n");
}

void UnitTestSCurve() {
    UnitTestSCurveCreateCloneFree();
    UnitTestSCurveLoadSavePrint();
    UnitTestSCurveGetSetCtrl();
    UnitTestSCurveGetAddRemoveSeg();
    UnitTestSCurveGet();
    UnitTestSCurveGetOrderDimNbSegMaxUNbCtrl();
    UnitTestSCurveGetApproxLenCenter();
    UnitTestSCurveRot();
    UnitTestSCurveScale();
    UnitTestSCurveTranslate();
    UnitTestSCurveGetBoundingBox();
    printf("UnitTestSCurve OK\n");
}

void UnitTestAll() {
    UnitTestBCurve();
    UnitTestSCurve();
    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
UnitTestBCurveGetOrderDim 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) <<0.000,1.000>> <2.000,3.000> <4.000,5.000> <<6.000,7.000>> <8.000,9.000> <10.000,11.000> <
UnitTestSCurveLoadSavePrint OK
UnitTestSCurveGetSetCtrl OK
UnitTestSCurveGetAddRemoveSeg OK
UnitTestSCurveGet OK
UnitTestSCurveGetOrderDimNbSegMaxUNbCtrl OK
UnitTestSCurveGetApproxLenCenter OK
UnitTestSCurveRot OK
UnitTestSCurveScale OK
UnitTestSCurveTranslate OK
UnitTestSCurveGetBoundingBox OK
UnitTestSCurve OK
UnitTestAll OK

```

bcurve.txt:

```

3 2
2 0.000000 1.000000
2 2.000000 3.000000
2 4.000000 5.000000
2 6.000000 7.000000

```

scurve.txt:

```

3 2 3
2 0.000000 1.000000
2 2.000000 3.000000
2 4.000000 5.000000
2 6.000000 7.000000
2 8.000000 9.000000
2 10.000000 11.000000
2 12.000000 13.000000
2 14.000000 15.000000
2 16.000000 17.000000
2 18.000000 19.000000

```