BCurve

P. Baillehache

October 30, 2017

Contents

1 Definitions **2**

1.1 BCurve definition 2

1.2 BCurve from cloud points 2

2 Interface **3**

3 Code **6**

4 Makefile **21**

5 Usage **21**

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 simply a GSet¡BCurve¿ to manipulate a set of curves.

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 $\overrightarrow{C_i}$ (vectors of dimension D). The curve in dimension D associated to the BCurve B is defined by $\overrightarrow{B(t)}$:

$$\begin{cases} \overrightarrow{B(t)} = \sum_{i=0}^{O} W_i^O(t)\overrightarrow{C_i} \text{ if } t \in [0.0, 1.0] \\ \overrightarrow{B(t)} = \overrightarrow{C_0} \text{ if } t < 0.0 \\ \overrightarrow{B(t)} = \overrightarrow{C_O} \text{ if } t > 1.0 \end{cases} \tag{1}$$

where, if $O = 0$

$$W_0^0(t) = 1.0 \tag{2}$$

and if $O \neq 0$

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

1.2 BCurve from cloud points

Given the cloud points made of N points $\overrightarrow{P_i}$, the BCurve of order $N-1$ passing through the N points (in the same order $\overrightarrow{P_0}$,$\overrightarrow{P_1}$,$\overrightarrow{P_2}$,... as given in input) can be obtained as follow.

If $N = 1$ the solution is trivial: $\overrightarrow{C_0} = \overrightarrow{P_0}$. As well, if $N = 2$ the solution is trivial: $\overrightarrow{C_0} = \overrightarrow{P_0}$ and $\overrightarrow{C_1} = \overrightarrow{P_1}$.

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

$$t_i = \frac{L(\overrightarrow{P_i})}{L(\overrightarrow{P_{N-1}})} \tag{4}$$

where

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

then we can calculate the C_i as follow. We have $\overrightarrow{C_0} = \overrightarrow{P_0}$ and $\overrightarrow{C_{N-1}} = \overrightarrow{P_{N-1}}$, and others $\overrightarrow{C_i}$ can be obtained by solving the linear system below for each dimension:

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

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 "pbmath.h"
#include "gset.h"

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

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

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

typedef struct SCurve {
  // Dimension
  int _dim;
  // Set of BCurve
  GSet *_curves;
} SCurve;

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

// Create a new BCurve of order 'order' and dimension 'dim'
// Return NULL if we couldn't create the BCurve
BCurve* BCurveCreate(int order, int dim);
```

```
// Clone the BCurve
// Return NULL if we couldn’t 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 0 in case of success, or:
// 1: invalid arguments
// 2: can’t allocate memory
// 3: invalid data
// 4: fscanf error
// 5: VecLoad error
int BCurveLoad(BCurve **that, FILE *stream);

// Save the BCurve to the stream
// Return 0 upon success, else
// 1: invalid arguments
// 2: fprintf error
// 3: VecSave error
int BCurveSave(BCurve *that, FILE *stream);

// Free the memory used by a BCurve
// Do nothing if arguments are invalid
void BCurveFree(BCurve **that);

// Print the BCurve on ’stream’
// Do nothing if arguments are invalid
void BCurvePrint(BCurve *that, FILE *stream);

// Set the value of the iCtrl-th control point to v
// Do nothing if arguments are invalid
void BCurveSet(BCurve *that, int iCtrl, VecFloat *v);

// Get the value of the BCurve at paramater ’u’ (in [0.0, 1.0])
// Return NULL if arguments are invalid or malloc failed
// if ’u’ < 0.0 it is replaced by 0.0
// if ’u’ > 1.0 it is replaced by 1.0
VecFloat* BCurveGet(BCurve *that, float u);

// Get the order of the BCurve
// Return -1 if argument is invalid
int BCurveOrder(BCurve *that);

// Get the dimension of the BCurve
// Return 0 if argument is invalid
int BCurveDim(BCurve *that);

// Get the approximate length of the BCurve (sum of dist between
// control points)
// Return 0.0 if argument is invalid
float BCurveApproxLen(BCurve *that);

// Rotate the curve CCW by ’theta’ radians relatively to the origin
// Do nothing if arguments are invalid
void BCurveRot2D(BCurve *that, float theta);

// Scale the curve by ’v’ relatively to the origin
// Do nothing if arguments are invalid
void BCurveScale(BCurve *that, VecFloat *v);

// Translate the curve by ’v’
// Do nothing if arguments are invalid
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 or the arguments are
// invalid
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])
// Return null if the arguments are invalid or memory allocation failed
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.
// Return NULL if arguments are invalid.
Shapoid* BCurveGetBoundingBox(BCurve *that);

// Create a new SCurve of dimension 'dim'
// Return NULL if we couldn't create the SCurve
SCurve* SCurveCreate(int dim);

// Clone the SCurve
// Return NULL if we couldn't 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 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
// 5: BCurveLoad error
int SCurveLoad(SCurve **that, FILE *stream);

// Save the SCurve to the stream
// Return 0 upon success, else
// 1: invalid arguments
// 2: fprintf error
// 3: BCurveSave error
int SCurveSave(SCurve *that, FILE *stream);

// Free the memory used by a SCurve
// Do nothing if arguments are invalid
void SCurveFree(SCurve **that);

// Print the SCurve on 'stream'
// Do nothing if arguments are invalid
void SCurvePrint(SCurve *that, FILE *stream);

// Set the 'iCurve'-th BCurve to a clone of 'curve'
// 'iCurve' must be in [0, current number of BCurve]
// 'curve' 's dimension must be equal to SCurve's dimension
// Do nothing if arguments are invalid
```

```
void SCurveSet(SCurve *that, int iCurve, BCurve *curve);

// Append a clone of 'curve'
// 'curve' 's dimension must be equal to SCurve's dimension
// Do nothing if arguments are invalid
void SCurveAdd(SCurve *that, BCurve *curve);

// Remove the 'iCurve'-th BCurve from the SCurve
// Return NULL if arguments are invalid
BCurve* SCurveRemove(SCurve *that, int iCurve);

// Get the 'iCurve'-th BCurve of the SCurve without removing it
// Return NULL if arguments are invalid
BCurve* SCurveGet(SCurve *that, int iCurve);

// Get the number of BCurve in the SCurve
// Return 0 if arguments are invalid
int SCurveGetNbCurve(SCurve *that);

// Get the dimension of the SCurve
// Return 0 if argument is invalid
int SCurveDim(SCurve *that);

// Get the approximate length of the SCurve (sum of approxLen
// of its BCurves)
// Return 0.0 if argument is invalid
float SCurveApproxLen(SCurve *that);

// Rotate the SCurve CCW by 'theta' radians relatively to the origin
// Do nothing if arguments are invalid
void SCurveRot2D(SCurve *that, float theta);

// Scale the SCurve by 'v' relatively to the origin
// Do nothing if arguments are invalid
void SCurveScale(SCurve *that, VecFloat *v);

// Translate the SCurve by 'v'
// Do nothing if arguments are invalid
void SCurveTranslate(SCurve *that, VecFloat *v);

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

#endif
```

3 Code

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

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

#include "bcurve.h"

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

// ================ Functions implementation ====================
// Create a new BCurve of order 'order' and dimension 'dim'
// Return NULL if we couldn't create the BCurve
BCurve* BCurveCreate(int order, int dim) {
  // Check arguments
  if (order < 0 || dim < 1)
    return NULL;
  // Allocate memory
  BCurve *that = (BCurve*)malloc(sizeof(BCurve));
  //If we could allocate memory
  if (that != NULL) {
    // Set the values
    that->_dim = dim;
    that->_order = order;
    // Allocate memory for the array of control points
    that->_ctrl = (VecFloat**)malloc(sizeof(VecFloat*) * (order + 1));
    // If we couldn't allocate memory
    if (that->_ctrl == NULL) {
      // Free memory
      free(that);
      // Stop here
      return NULL;
    }
    // For each control point
    for (int iCtrl = 0; iCtrl < order + 1; ++iCtrl) {
      // Allocate memory
      that->_ctrl[iCtrl] = VecFloatCreate(dim);
      // If we couldn't allocate memory
      if (that->_ctrl[iCtrl] == NULL) {
        // Free memory
        BCurveFree(&that);
        // Stop here
        return NULL;
      }
    }
  }
  // Return the new BCurve
  return that;
}

// Clone the BCurve
// Return NULL if we couldn't clone the BCurve
BCurve* BCurveClone(BCurve *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Allocate memory for the clone
  BCurve *clone = (BCurve*)malloc(sizeof(BCurve));
  // If we could allocate memory
  if (clone != NULL) {
    // Clone the properties
    clone->_dim = that->_dim;
    clone->_order = that->_order;
    // Allocate memory for the array of control points
    clone->_ctrl = (VecFloat**)malloc(sizeof(VecFloat*) *
      (clone->_order + 1));
    // If we couldn't allocate memory
    if (that->_ctrl == NULL) {
      // Free memory
      free(clone);
      // Stop here
      return NULL;
    }
    // For each control point
    for (int iCtrl = 0; iCtrl < clone->_order + 1; ++iCtrl) {
      // Clone the control point
      clone->_ctrl[iCtrl] = VecClone(that->_ctrl[iCtrl]);
      // If we couldn't clone the control point
      if (clone->_ctrl[iCtrl] == NULL) {
        // Free memory
        BCurveFree(&clone);
        // Stop here
        return NULL;
      }
    }
  }
  // Return the clone
  return clone;
}

// Load the BCurve from the stream
// If the BCurve is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
// 5: VecLoad error
int BCurveLoad(BCurve **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // 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 4;
  }
  // Allocate memory
  *that = BCurveCreate(order, dim);
  // If we coudln't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // 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 conrtol point
    // is not of the correct dimension
    if (ret != 0 || VecDim((*that)->_ctrl[iCtrl]) != (*that)->_dim) {
      // Free memory
      BCurveFree(that);
      // Stop here
      return 5;
    }
  }
  // Return success code
  return 0;
}

// Save the BCurve to the stream
// Return 0 upon success, or
// 1: invalid arguments
// 2: fprintf error
// 3: VecSave error
int BCurveSave(BCurve *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // 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 2;
  }
  // 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 != 0) {
      // Stop here
      return 3;
    }
  }
  // Return success code
  return 0;
}

// Free the memory used by a BCurve
// Do nothing if arguments are invalid
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 = 0; 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'
// Do nothing if arguments are invalid
void BCurvePrint(BCurve *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return;
  // 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);
    fprintf(stream, " ");
  }
}

// Set the value of the iCtrl-th control point to v
// Do nothing if arguments are invalid
void BCurveSet(BCurve *that, int iCtrl, VecFloat *v) {
  // Check arguments
  if (that == NULL || v == NULL || iCtrl < 0 ||
    iCtrl > that->_order || VecDim(v) != BCurveDim(that))
    return;
  // Set the values
  VecCopy(that->_ctrl[iCtrl], v);
}

// Get the value of the BCurve at paramater 'u' (in [0.0, 1.0])
// Return NULL if arguments are invalid or malloc failed
// if 'u' < 0.0 it is replaced by 0.0
// if 'u' > 1.0 it is replaced by 1.0
VecFloat* BCurveGet(BCurve *that, float u) {
  // Check arguments
  if (that == NULL)
    return NULL;
  if (u < 0.0)
    u = 0.0;
  if (u > 1.0)
    u = 1.0;
  // Allocate memory for the result
  VecFloat *v = VecFloatCreate(that->_dim);
  // If we couldn't allocate memory
  if (v == NULL)
    return NULL;
  // Declare a variable for calcul
  float *val = (float*)malloc(sizeof(float) * (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)
      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) {
        val[order] = (1.0 - u) * val[order] + u * val[order + 1];
      }
      --subOrder;
    }
    // Set the value for the current dim
    VecSet(v, dim, val[0]);
  }
  // Free memory
  free(val);
  // Return the result
  return v;
}
```

```
// Get the order of the BCurve
// Return -1 if argument is invalid
int BCurveOrder(BCurve *that) {
  // Check arguments
  if (that == NULL)
    return -1;
  return that->_order;
}

// Get the dimension of the BCurve
// Return 0 if argument is invalid
int BCurveDim(BCurve *that) {
  // Check arguments
  if (that == NULL)
    return 0;
  return that->_dim;
}

// Get the approximate length of the BCurve (sum of dist between
// control points)
// Return 0.0 if argument is invalid
float BCurveApproxLen(BCurve *that) {
  // Check arguments
  if (that == NULL)
    return 0.0;
  // Declare a variable to calculate the length
  float res = 0.0;
  // Calculate the length
  for (int iCtrl = 0; iCtrl < that->_order; ++iCtrl)
    res += VecDist(that->_ctrl[iCtrl], that->_ctrl[iCtrl + 1]);
  // Return the length
  return res;
}

// Rotate the curve CCW by 'theta' radians relatively to the origin
// Do nothing if arguments are invalid
void BCurveRot2D(BCurve *that, float theta) {
  // Check arguments
  if (that == NULL || that->_dim != 2)
    return;
  // For each control point
  for (int iCtrl = 0; iCtrl <= that->_order; ++iCtrl)
    // Rotate the control point
    VecRot2D(that->_ctrl[iCtrl], theta);
}

// Scale the curve by 'v' relatively to the origin
// Do nothing if arguments are invalid
void BCurveScale(BCurve *that, VecFloat *v) {
  // Check arguments
  if (that == NULL || v == NULL)
    return;
  // For each control point
  for (int iCtrl = 0; iCtrl <= that->_order; ++iCtrl)
    // Scale the control point
    for (int dim = 0; dim < VecDim(that->_ctrl[iCtrl]); ++dim)
      VecSet(that->_ctrl[iCtrl], dim,
        VecGet(that->_ctrl[iCtrl], dim) * VecGet(v, dim));
}

// Translate the curve by 'v'
// Do nothing if arguments are invalid
```

```
void BCurveTranslate(BCurve *that, VecFloat *v) {
  // Check arguments
  if (that == NULL || v == NULL)
    return;
  // For each control point
  for (int iCtrl = 0; iCtrl <= that->_order; ++iCtrl)
    // Translate the control point
    VecOp(that->_ctrl[iCtrl], 1.0, v, 1.0);
}

// 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 or the arguments are
// invalid
BCurve* BCurveFromCloudPoint(GSet *set) {
  // Check arguments
  if (set == NULL || set->_nbElem < 1)
    return NULL;
  // Declare a variable to memorize the result
  int order = set->_nbElem - 1;
  int dim = VecDim((VecFloat*)(set->_head->_data));
  BCurve *curve = BCurveCreate(order, dim);
  // If we could allocate memory
  if (curve != NULL) {
    // Set the first control point to the first point in the point cloud
    BCurveSet(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
      BCurveSet(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
        VecShort *dimMat = VecShortCreate(2);
        // Declare a variable to memorize the t values
        VecFloat *t = VecFloatCreate(order + 1);
        // If we could allocate memory
        if (t != NULL && dimMat != NULL) {
          // 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(VecGet(dimMat, 0));
            // If we could allocate memory
            if (m != NULL && v != NULL) {
              // 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));
                // If we could get the weights
                if (weight != NULL) {
                  // 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
              EqLinSys *sys = EqLinSysCreate(m, v);
              // If we could allocate memory
              if (sys != NULL) {
                // Solve the system
                VecFloat *solSys = EqLinSysSolve(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);
                }
              }
              // Free memory
              EqLinSysFree(&sys);
```

```
              VecFree(&v);
              MatFree(&m);
            }
          }
        }
        // Free memory
        VecFree(&dimMat);
        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])
// Return null if the arguments are invalid or memory allocation failed
VecFloat* BCurveGetWeightCtrlPt(BCurve *that, float t) {
  // Check arguments
  if (that == NULL || t < 0.0 || t > 1.0)
    return NULL;
  // Declare a variable to memorize the result
  VecFloat *res = VecFloatCreate(that->_order + 1);
  // If we could allocate memory
  if (res != NULL) {
    // 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--;) {
        // Calculate the weight of this control point
        // VecGet(v, - 1) = 0.0 and VecFloat is initialized to 0.0
        // => no need to check for border cases
        VecSet(res, iCtrl,
          (1.0 - t) * VecGet(res, iCtrl) + t * VecGet(res, iCtrl - 1));
      }
    }
  }
  // Return the result
  return res;
}

// Get the bounding box of the BCurve.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
// Return NULL if arguments are invalid.
Shapoid* BCurveGetBoundingBox(BCurve *that) {
  // Check argument
  if (that == NULL)
    return NULL;
  // Declare a variable to memorize the result
  Shapoid *res = FacoidCreate(that->_dim);
  // If we could allocate memory
  if (res != NULL) {
    // 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->_pos, iDim, VecGet(that->_ctrl[iCtrl], iDim));
          VecSet(res->_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->_pos, iDim))
            VecSet(res->_pos, iDim, VecGet(that->_ctrl[iCtrl], iDim));
          if (VecGet(that->_ctrl[iCtrl], iDim) >
            VecGet(res->_axis[iDim], iDim))
            VecSet(res->_axis[iDim], iDim,
              VecGet(that->_ctrl[iCtrl], iDim));
        }
      }
      VecSet(res->_axis[iDim], iDim,
        VecGet(res->_axis[iDim], iDim) - VecGet(res->_pos, iDim));
    }
  }
  // Return the result
  return res;
}

// Create a new SCurve of dimension ’dim’
// Return NULL if we couldn’t create the SCurve
SCurve* SCurveCreate(int dim) {
  // Check arguments
  if (dim <= 0)
    return NULL;
  // Declare a variable for the returned SCurve
  SCurve *ret = (SCurve*)malloc(sizeof(SCurve));
  // If we could allocate memory
  if (ret != NULL) {
    // Set the properties
    ret->_dim = dim;
    // Create the set
    ret->_curves = GSetCreate();
    // If we couldn’t allocate memory
    if (ret->_curves == NULL) {
      // Free memory and stop here
      SCurveFree(&ret);
      return NULL;
    }
  }
  // Return the new SCurve
  return ret;
}

// Clone the SCurve
// Return NULL if we couldn’t clone the SCurve
SCurve* SCurveClone(SCurve *that) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Allocate memory
  SCurve *ret = SCurveCreate(SCurveDim(that));
  // If we could allocate memory
  if (ret != NULL) {
    // Declare a pointer to the elements of the set
    GSetElem *ptr = that->_curves->_head;
    // Loop on elements
    while (ptr != NULL) {
      // Clone the BCurve and add it to the clone of SCurve
      GSetAppend(ret->_curves, BCurveClone((BCurve*)(ptr->_data)));
      // Move to the next element
      ptr = ptr->_next;
    }
  }
  // Return the cloned SCurve
  return ret;
}

// Load the SCurve from the stream
// If the SCurve is already allocated, it is freed before loading
// Return 0 in case of success, or:
// 1: invalid arguments
// 2: can't allocate memory
// 3: invalid data
// 4: fscanf error
// 5: BCurveLoad error
int SCurveLoad(SCurve **that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // If 'that' is already allocated
  if (*that != NULL) {
    // Free memory
    SCurveFree(that);
  }
  // Read the dimension and number of curve
  int dim = 0;
  int nbCurve = 0;
  int ret = fscanf(stream, "%d %d", &dim, &nbCurve);
  // If we couldn't read
  if (ret == EOF) {
    return 4;
  }
  // Allocate memory
  *that = SCurveCreate(dim);
  // If we couldn't allocate memory
  if (*that == NULL) {
    return 2;
  }
  // Loop on curves
  for (int iCurve = 0; iCurve < nbCurve; ++iCurve) {
    // Declare a variable to load the BCurve
    BCurve *curve = NULL;
    // Load the BCurve
    ret = BCurveLoad(&curve, stream);
    // If we couldn't load the BCurve
    if (ret != 0)
      return 5;
    // Check the dimension of the curve
    if (BCurveDim(curve) != dim)
      return 3;
    // Add the BCurve to the SCurve
    SCurveAdd(*that, curve);
  }
  return 0;
}
```

```
// Save the SCurve to the stream
// Return 0 upon success, else
// 1: invalid arguments
// 2: fprintf error
// 3: BCurveSave error
int SCurveSave(SCurve *that, FILE *stream) {
  // Check arguments
  if (that == NULL || stream == NULL)
    return 1;
  // Save the dimension and number of curve
  int ret = fprintf(stream, "%d %d\n", that->_dim,
    that->_curves->_nbElem);
  // If the fprintf failed
  if (ret < 0)
    // Stop here
    return 2;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = that->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Save the BCurve
    BCurveSave((BCurve*)(ptr->_data), stream);
    // Move to the next BCurve
    ptr = ptr->_next;
  }
  return 0;
}

// Free the memory used by a SCurve
// Do nothing if arguments are invalid
void SCurveFree(SCurve **that) {
  // Check argument
  if (that == NULL || *that == NULL)
    return;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = (*that)->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Free the BCurve
    BCurveFree((BCurve**)(&(ptr->_data)));
    // Move to the next BCurve
    ptr = ptr->_next;
  }
  // Free memory
  GSetFree(&((*that)->_curves));
  free(*that);
  *that = NULL;
}

// Print the SCurve on 'stream'
// Do nothing if arguments are invalid
void SCurvePrint(SCurve *that, FILE *stream) {
  // Check argument
  if (that == NULL || stream == NULL)
    return;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = that->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Print the BCurve
    BCurvePrint((BCurve*)(ptr->_data), stream);
    fprintf(stream, "\n");
    // Move to the next BCurve
    ptr = ptr->_next;
  }
}

// Set the 'iCurve'-th BCurve to a clone of 'curve'
// 'iCurve' must be in [0, current number of BCurve]
// 'curve' 's dimension must be equal to SCurve's dimension
// Do nothing if arguments are invalid
void SCurveSet(SCurve *that, int iCurve, BCurve *curve) {
  // Check arguments
  if (that == NULL || curve == NULL || iCurve < 0 ||
    iCurve > that->_curves->_nbElem)
    return;
  // Clone the curve
  BCurve *clone = BCurveClone(curve);
  // If we could clone)
  if (clone != NULL)
    // Insert a clone of the curve
    GSetInsert(that->_curves, clone, iCurve);
}

// Append a clone of 'curve'
// 'curve' 's dimension must be equal to SCurve's dimension
// Do nothing if arguments are invalid
void SCurveAdd(SCurve *that, BCurve *curve) {
  // Check arguments
  if (that == NULL || curve == NULL)
    return;
  // Append the curve
  SCurveSet(that, that->_curves->_nbElem, curve);
}

// Remove the 'iCurve'-th BCurve from the SCurve
// Return NULL if arguments are invalid
BCurve* SCurveRemove(SCurve *that, int iCurve) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Get the BCurve out of the set
  BCurve *curve = (BCurve*)GSetRemove(that->_curves, iCurve);
  // Return the curve
  return curve;
}

// Get the 'iCurve'-th BCurve of the SCurve, without removing it
// Return NULL if arguments are invalid
BCurve* SCurveGet(SCurve *that, int iCurve) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Return the BCurve
  return (BCurve*)(GSetGet(that->_curves, iCurve));
}

// Get the number of BCurve in the SCurve
// Return 0 if arguments are invalid
int SCurveGetNbCurve(SCurve *that) {
  // Check arguments
  if (that == NULL)
    return 0;
  // Return the number of BCurves
  return that->_curves->_nbElem;
}

// Get the dimension of the SCurve
// Return 0 if argument is invalid
int SCurveDim(SCurve *that) {
  // Check arguments
  if (that == NULL)
    return 0;
  // Return the dimension
  return that->_dim;
}

// Get the approximate length of the SCurve (sum of approxLen
// of its BCurves)
// Return 0.0 if argument is invalid
float SCurveApproxLen(SCurve *that) {
  // Check arguments
  if (that == NULL)
    return 0.0;
  // Declare a variable to calculate the length
  float length = 0.0;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = that->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Add the approximate length of this BCurve
    length += BCurveApproxLen((BCurve*)(ptr->_data));
    // Move to the next BCurve
    ptr = ptr->_next;
  }
  // Return the length
  return length;
}

// Rotate the SCurve CCW by 'theta' radians relatively to the origin
// Do nothing if arguments are invalid
void SCurveRot2D(SCurve *that, float theta) {
  // Check arguments
  if (that == NULL)
    return;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = that->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Rotate the BCurve
    BCurveRot2D((BCurve*)(ptr->_data), theta);
    // Move to the next BCurve
    ptr = ptr->_next;
  }
}

// Scale the SCurve by 'v' relatively to the origin
// Do nothing if arguments are invalid
void SCurveScale(SCurve *that, VecFloat *v) {
  // Check arguments
  if (that == NULL || v == NULL)
    return;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = that->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Rotate the BCurve
    BCurveScale((BCurve*)(ptr->_data), v);
    // Move to the next BCurve
    ptr = ptr->_next;
  }
}

// Translate the SCurve by 'v'
// Do nothing if arguments are invalid
void SCurveTranslate(SCurve *that, VecFloat *v) {
  // Check arguments
  if (that == NULL || v == NULL)
    return;
  // Declare a pointer on elements of the set of curves
  GSetElem *ptr = that->_curves->_head;
  // Loop on elements
  while (ptr != NULL) {
    // Translate the BCurve
    BCurveTranslate((BCurve*)(ptr->_data), v);
    // Move to the next BCurve
    ptr = ptr->_next;
  }
}

// Get the bounding box of the SCurve.
// Return a Facoid whose axis are aligned on the standard coordinate
// system.
// Return NULL if arguments are invalid.
Shapoid* SCurveGetBoundingBox(SCurve *that) {
  // Check arguments
  if (that == NULL)
    return NULL;
  // Allocate memory for the set of bounding boxes of BCurve
  GSet *set = GSetCreate();
  // If we couldn't allocate memory
  if (set == NULL) {
    return NULL;
  }
  // Add the bounding box of each BCurve
  GSetElem *ptr = set->_head;
  while (ptr != NULL) {
    GSetAppend(set, BCurveGetBoundingBox((BCurve*)(ptr->_data)));
    ptr = ptr->_next;
  }
  // Get the bounding box of the set of bounding boxes of BCurve
  Shapoid *ret = ShapoidGetBoundingBoxSet(set);
  // Free memory used by the set of bounding boxes of BCurve
  ptr = set->_head;
  while (ptr != NULL) {
    ShapoidFree((Shapoid**)(&(ptr->_data)));
    ptr = ptr->_next;
  }
  GSetFree(&set);
  // Return the result
  return ret;
}
```

4 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
OPTIONS_RELEASE=-O3
OPTIONS=$(OPTIONS_RELEASE)
INCPATH=/home/bayashi/Coding/Include
LIBPATH=/home/bayashi/Coding/Include

all : main

main: main.o bcurve.o $(LIBPATH)/pbmath.o $(LIBPATH)/gset.o Makefile
gcc $(OPTIONS) main.o bcurve.o $(LIBPATH)/pbmath.o $(LIBPATH)/gset.o -o main -lm

main.o : main.c bcurve.h Makefile
gcc $(OPTIONS) -I$(INCPATH) -c main.c

bcurve.o : bcurve.c bcurve.h $(INCPATH)/pbmath.h $(INCPATH)/gset.h Makefile
gcc $(OPTIONS) -I$(INCPATH) -c bcurve.c

clean :
rm -rf *.o main

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

install :
cp bcurve.h ../Include; cp bcurve.o ../Include
```

5 Usage

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

float CloudCurveX(float t) {
  return t * t;
  //return 2.0 * cos(t * PBMATH_HALFPI);
}

float CloudCurveY(float t) {
  return sqrt(t);
  //return sin(t * PBMATH_HALFPI);
}

int main(int argc, char **argv) {
  // Create a BCurve
  int order = 3;
  int dim = 2;
  BCurve *curve = BCurveCreate(order, dim);
  // If we couldn't create the BCurve
  if (curve == NULL) {
    // Print a message
    fprintf(stderr, "BCurveCreate failed\n");
    // Stop here
    return 1;
  }
```

```
// Print the BCurve
BCurvePrint(curve, stdout);
fprintf(stdout, "\n");
// Create a VecFloat to set the values
VecFloat *v = VecFloatCreate(dim);
// If we couldn’t create the VecFloat
if (v == NULL) {
  // Release memory
  BCurveFree(&curve);
  // Stop here
  return 2;
}
// Set the control points
float ctrlPts[8] = {0.0, 1.0, 2.0, 5.0, 4.0, 3.0, 6.0, 7.0};
for (int iCtrl = 0; iCtrl < order + 1; ++iCtrl) {
  VecSet(v, 0, ctrlPts[2 * iCtrl]);
  VecSet(v, 1, ctrlPts[2 * iCtrl + 1]);
  BCurveSet(curve, iCtrl, v);
}
// Print the BCurve
BCurvePrint(curve, stdout);
fprintf(stdout, "\n");
// Save the curve
FILE *file = fopen("./curve.txt", "w");
// If we couldn’t open the file
if (file == NULL) {
  // Print a message
  fprintf(stderr, "Can’t open file\n");
  // Free memory
  VecFree(&v);
  BCurveFree(&curve);
  // Stop here
  return 3;
}
int ret = BCurveSave(curve, file);
// If we couldn’t save
if (ret != 0) {
  // Print a message
  fprintf(stderr, "BCurveSave failed (%d)\n", ret);
  // Free memory
  VecFree(&v);
  BCurveFree(&curve);
  // Stop here
  return 4;
}
fclose(file);
// Load the curve
file = fopen("./curve.txt", "r");
// If we couldn’t open the file
if (file == NULL) {
  // Print a message
  fprintf(stderr, "Can’t open file\n");
  // Free memory
  VecFree(&v);
  BCurveFree(&curve);
  // Stop here
  return 5;
}
BCurve *loaded = NULL;
ret = BCurveLoad(&loaded, file);
// If we couldn’t load
if (ret != 0) {
  // Print a message
  fprintf(stderr, "BCurveLoad failed (%d)\n", ret);
  // Free memory
  VecFree(&v);
  BCurveFree(&curve);
  BCurveFree(&loaded);
  // Stop here
  return 6;
}
fclose(file);
// Print the loaded curve
BCurvePrint(loaded, stdout);
fprintf(stdout, "\n");
// Get some values of the curve
for (float u = 0.0; u <= 1.01; u += 0.1) {
  VecFloat *w = BCurveGet(curve, u);
  // If we couldn't get the values
  if (w == NULL) {
    // Free memory
    VecFree(&v);
    BCurveFree(&curve);
    BCurveFree(&loaded);
    // Stop here
    return 7;
  }
  fprintf(stdout, "%.1f: ", u);
  VecPrint(w, stdout);
  fprintf(stdout, "\n");
  VecFree(&w);
}
// Scale the curve
VecSet(v, 0, 0.5);
VecSet(v, 1, 1.0);
BCurveScale(curve, v);
// Rotate the curve
BCurveRot2D(curve, PBMATH_PI * 0.5);
// Translate the curve
VecSet(v, 0, -0.5);
VecSet(v, 1, 1.0);
BCurveTranslate(curve, v);
// Get some values of the curve
fprintf(stdout, "After transformation:\n");
for (float u = 0.0; u <= 1.01; u += 0.1) {
  VecFloat *w = BCurveGet(curve, u);
  // If we couldn't get the values
  if (w == NULL) {
    // Free memory
    VecFree(&v);
    BCurveFree(&curve);
    BCurveFree(&loaded);
    // Stop here
    return 7;
  }
  fprintf(stdout, "%.1f: ", u);
  VecPrint(w, stdout);
  fprintf(stdout, "\n");
  VecFree(&w);
}
// Print the curve approximate length
fprintf(stdout, "approx length: %.3f\n", BCurveApproxLen(curve));
// Print the weight of control points
fprintf(stdout, "Control points weight:\n");
```

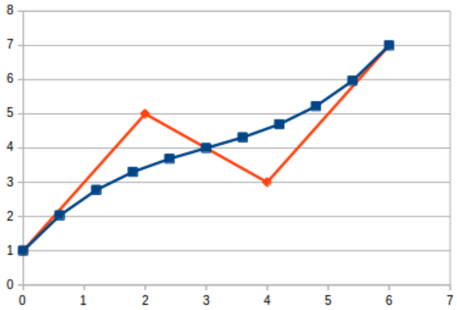
```
for (float t = 0.0; t <= 1.01; t += 0.05) {
  if (t > 1.0) t = 1.0;
  VecFloat *w = BCurveGetWeightCtrlPt(curve, t);
  if (w != NULL) {
    fprintf(stdout, "%.3f ", t);
    VecPrint(w, stdout);
    fprintf(stdout, "\n");
  }
  VecFree(&w);
}
// Get a curve from a cloud point
GSet *cloud = GSetCreate();
if (cloud != NULL) {
  VecFloat *w = NULL;
  fprintf(stdout, "cloud:\n");
  //for (float t = 0.0; t < 1.01; t += 0.25) {
  //for (float t = 0.0; t < 1.01; t += 0.334) {
  for (float t = 0.0; t < 1.01; t += 0.5) {
    w = VecFloatCreate(2);
    GSetAppend(cloud, w);
    VecSet(w, 0, CloudCurveX(t));
    VecSet(w, 1, CloudCurveY(t));
    VecPrint(w, stdout);
    fprintf(stdout, "\n");
  }
  w = NULL;
  BCurve *cloudCurve = BCurveFromCloudPoint(cloud);
  if (cloudCurve == NULL) {
    fprintf(stdout, "Couldn't get curve from cloud\n");
    return 8;
  }
  fprintf(stdout, "cloudCurve: ");
  BCurvePrint(cloudCurve, stdout);
  fprintf(stdout, "\n");
  for (float t = 0.0; t < 1.01; t += 0.1) {
    if (t > 1.0) t = 1.0;
    fprintf(stdout, "%.3f ", t);
    w = BCurveGet(cloudCurve, t);
    VecPrint(w, stdout);
    fprintf(stdout, "\n");
    VecFree(&w);
  }
  BCurveFree(&cloudCurve);
}
// Get the bounding box of the curve
Shapoid *bound = BCurveGetBoundingBox(curve);
if (bound == NULL) {
  fprintf(stdout, "Couldn't get the bounding box\n");
  return 9;
}
fprintf(stdout, "bounding box of \n");
BCurvePrint(curve, stdout);
fprintf(stdout, "\nis\n");
ShapoidPrint(bound, stdout);
ShapoidFree(&bound);
// Free memory
GSetElem *elem = cloud->_head;
while (elem != NULL) {
  VecFree((VecFloat**)(&(elem->_data)));
  elem = elem->_next;
}
GSetFree(&cloud);
```

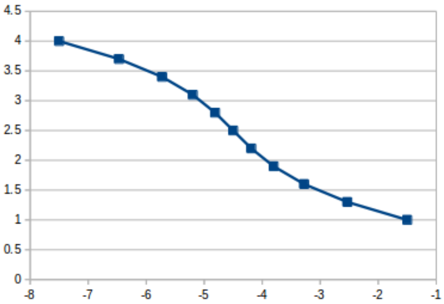
```
  VecFree(&v);
  BCurveFree(&curve);
  BCurveFree(&loaded);
  // Return success code
  return 0;
}
```

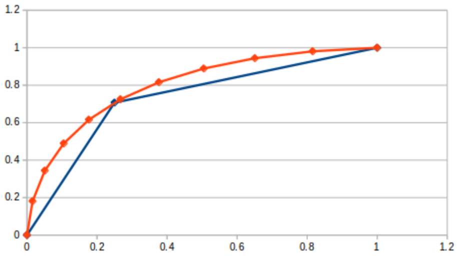
Output:

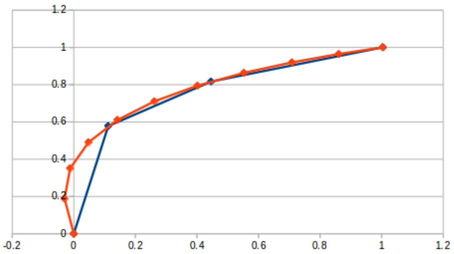
```
order(3) dim(2) <0.000,0.000> <0.000,0.000> <0.000,0.000> <0.000,0.000>
order(3) dim(2) <0.000,1.000> <2.000,5.000> <4.000,3.000> <6.000,7.000>
order(3) dim(2) <0.000,1.000> <2.000,5.000> <4.000,3.000> <6.000,7.000>
0.0: <0.000,1.000>
0.1: <0.600,2.032>
0.2: <1.200,2.776>
0.3: <1.800,3.304>
0.4: <2.400,3.688>
0.5: <3.000,4.000>
0.6: <3.600,4.312>
0.7: <4.200,4.696>
0.8: <4.800,5.224>
0.9: <5.400,5.968>
1.0: <6.000,7.000>
After transformation:
0.0: <-1.500,1.000>
0.1: <-2.532,1.300>
0.2: <-3.276,1.600>
0.3: <-3.804,1.900>
0.4: <-4.188,2.200>
0.5: <-4.500,2.500>
0.6: <-4.812,2.800>
0.7: <-5.196,3.100>
0.8: <-5.724,3.400>
0.9: <-6.468,3.700>
1.0: <-7.500,4.000>
approx length: 10.482
Control points weight:
0.000 <1.000,0.000,0.000,0.000>
0.050 <0.857,0.135,0.007,0.000>
0.100 <0.729,0.243,0.027,0.001>
0.150 <0.614,0.325,0.057,0.003>
0.200 <0.512,0.384,0.096,0.008>
0.250 <0.422,0.422,0.141,0.016>
0.300 <0.343,0.441,0.189,0.027>
0.350 <0.275,0.444,0.239,0.043>
0.400 <0.216,0.432,0.288,0.064>
0.450 <0.166,0.408,0.334,0.091>
0.500 <0.125,0.375,0.375,0.125>
0.550 <0.091,0.334,0.408,0.166>
0.600 <0.064,0.288,0.432,0.216>
0.650 <0.043,0.239,0.444,0.275>
0.700 <0.027,0.189,0.441,0.343>
0.750 <0.016,0.141,0.422,0.422>
0.800 <0.008,0.096,0.384,0.512>
0.850 <0.003,0.057,0.325,0.614>
0.900 <0.001,0.027,0.243,0.729>
0.950 <0.000,0.007,0.135,0.857>
1.000 <0.000,0.000,0.000,1.000>
cloud:
<0.000,0.000>
```

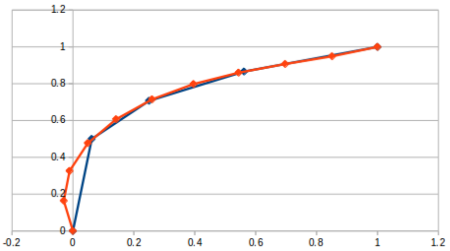
```
<0.250,0.707>
<1.000,1.000>
cloudCurve: order(2) dim(2) <0.000,0.000> <0.035,0.950> <1.000,1.000>
0.000 <0.000,0.000>
0.100 <0.016,0.181>
0.200 <0.051,0.344>
0.300 <0.105,0.489>
0.400 <0.177,0.616>
0.500 <0.267,0.725>
0.600 <0.377,0.816>
0.700 <0.505,0.889>
0.800 <0.651,0.944>
0.900 <0.816,0.981>
1.000 <1.000,1.000>
bounding box of
order(3) dim(2) <-1.500,1.000> <-5.500,2.000> <-3.500,3.000> <-7.500,4.000>
is
Type: Facoid
Dim: 2
Pos: <-7.500,1.000>
Axis(0): <6.000,0.000>
Axis(1): <0.000,3.000>
```











curve.txt:

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