

BCurve

P. Baillehache

September 28, 2017

Contents

1	Interface	1
2	Code	3
3	Makefile	7
4	Usage	8

Introduction

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

1 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"

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

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

```

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

// ===== 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 parameter '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);

#endif

```

2 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 couldn'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 control 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 parameter '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;
}

```

3 Makefile

```
OPTIONS_DEBUG=-ggdb -g3 -Wall
```

```

OPTIONS_RELEASE=-03
OPTIONS=$(OPTIONS_RELEASE)
INCPATH=/home/bayashi/Coding/Include
LIBPATH=/home/bayashi/Coding/Include

all : main

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

```

4 Usage

```

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

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);
}
// Print the curve approximate length
fprintf(stdout, "approx length: %.3f\n", BCurveApproxLen(curve));
// Free memory
VecFree(&v);
BCurveFree(&curve);
BCurveFree(&loaded);
// Return success code
return 0;
}

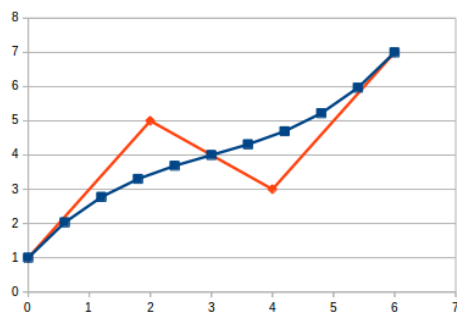
```

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>
approx length: 11.773

```



curve.txt:

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