

October 28, 2016 00:28

**1. Introduction.** 이 문서는 Gerald Farin의 “Curves and Surfaces for CAGD: A Practical Guide” 4<sup>th</sup> edition에 기술되어 있는 알고리즘들을 간략하게 설명하고 구현한다.

가장 먼저  $n$ -차원 유클리드 공간에 존재하는 점을 기술하기 위한 **point** 타입을 정의한다. 뒤에서 좀 더 자세하게 설명하겠지만, 유클리드 공간의 point는 위치를 나타내는 position vector와 다른 특성을 갖는다. 예를 들면, point 사이의 뺄셈은 정의되지만 덧셈은 물리적으로 의미가 성립되지 않아 정의될 수 없는 것을 들 수 있다.

두 번째로, 추상적인 타입으로 **curve** 타입을 정의한다. 이 타입은 일반적인 곡선에서 필요로 하는 몇 가지 인터페이스를 정의하고, PostScript 파일 출력을 위한 method들을 갖는다.

**curve** 타입을 base class로 Bézier 곡선을 기술하기 위한 **bezier** 타입을 정의한다. 그리고 여러 개의 곡선들을 이어 붙여 사용하기 위한 **piecewise\_bezier\_curve** 타입을 정의한다. 마지막으로 가장 널리 쓰이는 cubic spline curve를 기술하기 위하여 **cubic\_spline** 타입을 정의한다.

**2. Namespace.** 이 문서에서 기술하는 모든 타입과 유틸리티 함수들은 **cagd** namespace에 정의한다. 연산 결과가 0과 유사할 때 0으로 판별하기 위하여 machine epsilon을  $2.2204 \cdot 10^{-16}$ 으로 정의한다.

**cagd** namespace 객체의 method들을 실행하는 도중 오류가 발생할 때에는 오류의 원인을 설명하고 오류의 종류를 구별할 수 있도록 오류 코드를 객체 내에 저장한다. 오류 코드를 정의하기 위하여 enumeration을 정의한다. 앞으로 다른 타입들과 그 각각의 method들을 정의하면서 그것들과 연관된 오류 코드들도 추가로 정의할 것이다. 매우 자명하게도, **NO\_ERR**는 method를 성공적으로 수행하고 아무런 오류가 없음을 의미한다.

점이나 곡선과 같은 기하학적 객체를 다룰 때 실행결과를 가장 쉽게 확인하는 방법은 그것들을 2차원 지면상에 실제로 그리는 것이다. 또한 그 결과를 편리하게 활용할 수 있도록 간단한 PostScript 출력을 지원하는 타입과 method들을 구현한다. 이 프로그램에서는 PostScript 파일을 가리키는 타입으로 **psf**를 정의한다. (실제로는 C++의 **ofstream** 타입에 다른 이름을 붙였을 뿐이다.)

OpenCL을 이용한 병렬연산을 수행할 수 있도록 **mpoi.h** 헤더를 추가한다.

`<cagd.h 2>` ≡

```

1  #ifndef __COMPUTER_AIDED_GEOMETRIC_DESIGN_H_
2  #define __COMPUTER_AIDED_GEOMETRIC_DESIGN_H_
3  #include <cstdint>
4  #include <vector>
5  #include <list>
6  #include <string>
7  #include <cstdlib>
8  #include <algorithm>
9  #include <cmath>
10 #include <iostream>
11 #include <ostream>
12 #include <fstream>
13 #include <sstream>
14 #include <ios>
15 #include <iterator>
16 #include <initializer_list>
17 #include "mpoi.h"
18 #if defined (_WIN32) ∨ defined (_WIN64)
19 #define NOMINMAX
20 #define M_PI 3.14159265358979323846
21 #define M_PI_2 1.57079632679489661923
22 #define M_PI_4 0.785398163397448309616
23 #endif
24 using namespace std;
25 namespace cagd {
26     const double EPS =  $2.2204 \cdot 10^{-16}$ ;
27     enum err_code {
28         <Error codes of cagd 34>
29         NO_ERR

```

```

30     };
31     typedef ofstream psf;
32     〈Definition of point 7〉
33     〈Definition of curve 27〉
34     〈Definition of bezier 42〉
35     〈Definition of piecewise_bezier_curve 74〉
36     〈Definition of cubic_spline 99〉
37     〈Declaration of cagd functions 5〉
38 }
39 #endif

```

### 3. Implementation of **cagd**.

〈**cagd.cpp** 3〉 ≡

```

40 #include "cagd.h"
41 using namespace cagd;
42 〈Implementation of cagd functions 4〉
43 〈Implementation of point 9〉
44 〈Implementation of curve 28〉
45 〈Implementation of bezier 44〉
46 〈Implementation of piecewise_bezier_curve 75〉
47 〈Implementation of cubic_spline 101〉

```

### 4. PostScript 파일을 생성하고 닫기 위한 함수를 정의한다.

〈Implementation of **cagd** functions 4〉 ≡

```

48 psf cagd::create_postscript_file(string file_name) {
49     psf ps_file;
50     ps_file.open(file_name.c_str(), ios_base::out);
51     if ( $\neg$ ps_file) {
52         exit(-1);
53     }
54     ps_file << "%!PS-Adobe-3.0" << endl << "/Helvetica findfont 10 scalefont setfont" << endl;
55     return ps_file;
56 }
57 void cagd::close_postscript_file(psf &ps_file, bool with_new_page) {
58     if (with_new_page ≡ true) {
59         ps_file << "showpage" << endl;
60     }
61     ps_file.close();
62 }

```

56, 67, 130, 133, 135, 137번 마디도 살펴보라.

이 코드는 3번 마디에서 사용된다.

5.    〈Declaration of **cagd** functions 5〉 ≡

```
63        psf create_postscript_file(string);  
64        void close_postscript_file(psf &, bool);
```

17, 25, 57, 68, 131, 134, 136, 141번 마디도 살펴보라.  
이 코드는 2번 마디에서 사용된다.

**6. Test program.**

```

<test.cpp 6> ≡
65 #include <iostream>
66 #include <iomanip>
67 #include <chrono>
68 #include "cagd.h"
69 using namespace cagd; using namespace std::chrono;
70 void print_title(const char *);
71 void print_title(const char *str) {
72     cout << endl << endl;
73     char prev = cout.fill(' ');
74     cout << ">>" << setw(68) << ' ' << endl;
75     cout << ">>" << endl;
76     cout << ">>TEST:" << str << endl;
77     cout << ">>" << endl;
78     cout << ">>" << setw(68) << ' ' << endl;
79     cout.fill(prev);
80 }
81 int main(int argc, char *argv[]) {
82     cout <<
83     "=====\\n" <<
84     "\\n" <<
85     "TEST: CAGD LIBRARY\\n" <<
86     "\\n" <<
87     "=====\\n\\n";
88     <Test routines 26>;
89     return 0;
90 }

```

**7. Point in Euclidean space.** `point` 타입은 일반적인  $n$ -차원 유클리드 공간에 존재하는 하나의 점을 기술한다.

〈Definition of `point` 7〉≡

```
91 struct point {
92     〈Data members of point 8〉
93     〈Methods of point 11〉
94 };
```

이 코드는 2번 마디에서 사용된다.

**8. `point` 타입의 data member는 아주 간단하다.**  $n$ -차원 유클리드 공간에 존재하는 점은  $n$ 개의 좌표를 저장한다.

〈Data members of `point` 8〉≡

```
95 vector<double> _elem;
```

이 코드는 7번 마디에서 사용된다.

**9. `point` 타입에 대하여 적용할 수 있는 method들은**

1. 객체에 대한 property들;
2. Constructor들과 destructor;
3. Assignment, 덧셈과 뺄셈, 상수배 등의 연산자들;
4. 점들 사이의 거리를 계산하는 등의 utility 함수들 이 있다.

〈Implementation of `point` 9〉≡

```
96     〈Properties of point 10〉
97     〈Constructors and destructor of point 12〉
98     〈Operators of point 14〉
99     〈Other member functions of point 20〉
100     〈Non-member functions for point 16〉
```

이 코드는 3번 마디에서 사용된다.

**10. `point` 타입의 객체가 갖는 property에 접근하기 위한 몇 가지 method들을 정의한다.** `dimension()`과 `dim()` method는 `point` 타입의 객체가 몇 차원 공간의 점인지 알려준다.

〈Properties of `point` 10〉≡

```
101 size_t point::dimension() const {
102     return (this->_elem).size();
103 }
104 size_t point::dim() const {
105     return (this->_elem).size();
106 }
```

이 코드는 9번 마디에서 사용된다.

11. 〈Methods of **point** 11〉 ≡

```
107     size_t dimension() const;
```

```
108     size_t dim() const;
```

13, 15, 19, 21, 23번 마디도 살펴보라.

이 코드는 7번 마디에서 사용된다.

12. **point** 타입의 constructor와 destructor를 정의한다.

- 아무런 argument가 주어지지 않는 경우 몇 차원의 **point** 객체를 생성해야 할지 알 수 없으므로, default constructor의 생성을 방지한다. (C++11 필요.)
- 복사 생성자 (copy constructor)는 직접 정의하고, 임의 갯수의 **double** 타입 인자를 받아 그 갯수만큼의 차원을 갖는 **point** 객체를 생성하기 위하여 **initializer\_list**를 이용한 생성자를 구현한다.
- 생성자의 인자로 단 하나의 정수  $n$ 만 주어지면, 모든 원소가 0인  $n$ -차원 **point** 객체를 생성한다.
- **double**타입의 배열로부터 **point** 객체를 생성하는 constructor를 정의한다.

〈Constructors and destructor of **point** 12〉 ≡

```
109     point::point(const point &src)
```

```
110         : _elem(src._elem) {}
```

```
111     point::point(initializer_list<double> v)
```

```
112         : _elem(vector<double>(v.begin(), v.end())) {}
```

```
113     point::point(const double v1, const double v2, const double v3)
```

```
114         : _elem(vector<double>(3))
```

```
115     {
```

```
116         _elem[0] = v1;
```

```
117         _elem[1] = v2;
```

```
118         _elem[2] = v3;
```

```
119     }
```

```
120     point::point(const size_t n)
```

```
121         : _elem(vector<double>(n, 0.)) {}
```

```
122     point::point(const size_t n, const double *v)
```

```
123         : _elem(vector<double>(n, 0.))
```

```
124     {
```

```
125         for (size_t i = 0; i < n; i++) {
```

```
126             _elem[i] = v[i];
```

```
127         }
```

```
128     }
```

```
129     point::~point() {}
```

이 코드는 9번 마디에서 사용된다.

**13.**  $\langle$ Methods of **point** 11 $\rangle + \equiv$ 

```
130   point() = delete ;
131   point(const point &);
132   point(initializer_list<double>);
133   point(const double, const double, const double  $v_3 = 0.$ );
134   point(const size_t);
135   point(const size_t, const double *);
136   virtual ~point();
```



**14. Operators of point.** **point** 타입 객체들 사이의 덧셈과 뺄셈, scalar와의 곱셈과 나눗셈을 위한 method 들을 정의한다. 덧셈과 뺄셈, scalar와의 곱셈, 나눗셈의 구현은 매우 자명하므로 설명은 생략한다. 나눗셈의 경우 젯수가 0이면 아무런 연산도 수행하지 않고 그대로 리턴한다.

〈Operators of **point** 14〉 ≡

```

137     void point::operator=(const point &src) {
138         _elem = src._elem;
139     }
140     point &point::operator*=(const double s) {
141         size_t sz = this->dim();
142         for (size_t i = 0; i < sz; i++) {
143             (this->elem[i]) *= s;
144         }
145         return *this;
146     }
147     point &point::operator/=(const double s) {
148         if (s == 0.) return *this;
149         size_t sz = this->dim();
150         for (size_t i = 0; i < sz; i++) {
151             (this->elem[i]) /= s;
152         }
153         return *this;
154     }
155     point &point::operator+=(const point &pt) {
156         size_t sz_min = min(this->dim(), pt.dim());
157         for (size_t i = 0; i < sz_min; i++) {
158             (this->elem[i]) += pt._elem[i];
159         }
160         return *this;
161     }
162     point &point::operator-=(const point &pt) {
163         size_t sz_min = min(this->dim(), pt.dim());
164         for (size_t i = 0; i < sz_min; i++) {
165             (this->elem[i]) -= pt._elem[i];
166         }
167         return *this;
168     }

```

18번 마디도 살펴보라.

이 코드는 9번 마디에서 사용된다.

15.  $\langle \text{Methods of point 11} \rangle + \equiv$ 

```

169 void operator=(const point &);
170 point &operator*=(const double);
171 point &operator/=(const double);
172 point &operator+=(const point &);
173 point &operator-=(const point &);

```

16. 몇 가지 이항연산자들과 단항연산자(negation)를 추가로 정의한다. 두 개의 **point** 타입 변수  $a$ 와  $b$ 에 대하여  $a + b$ 를 **operator+(point, point)** 함수 내에서 **return**  $pt1 += pt2$ 로 구현되어 있다고 해서  $a$ 가 바뀌는 것은 아니다. 이는 함수 호출의 convention이 call-by-value이기 때문에  $a$ 와  $b$ 가 각각  $pt1$ 와  $pt2$ 로 복사되기 때문이다. 따라서  $pt1$ 은 값이 바뀌지만 원래 expression을 구성하는  $a$ 는 바뀌지 않는다.

 $\langle \text{Non-member functions for point 16} \rangle \equiv$ 

```

174 point cagd::operator*(double s, point pt) {
175     return pt *= s;
176 }
177 point cagd::operator*(point pt, double s) {
178     return pt *= s;
179 }
180 point cagd::operator/(point pt, double s) {
181     return pt /= s;
182 }
183 point cagd::operator+(point pt1, point pt2) {
184     return pt1 += pt2;
185 }
186 point cagd::operator-(point pt1, point pt2) {
187     return pt1 -= pt2;
188 }
189 point cagd::operator-(point pt1) {
190     size_t sz = pt1.dim();
191     cagd::point negated(sz);
192     for (size_t i = 0; i < sz; i++) {
193         negated._elem[i] = -pt1._elem[i];
194     }
195     return negated;
196 }

```

24번 마디도 살펴보라.

이 코드는 9번 마디에서 사용된다.

17. 〈Declaration of **cagd** functions 5〉 +≡

```

197 point operator*(double, point);
198 point operator*(point, double);
199 point operator/(point, double);
200 point operator+(point, point);
201 point operator-(point, point);
202 point operator-(point);

```

18.  $n$ -차원 공간에 존재하는 **point** 타입 객체의  $i$  번째 좌표에 접근하기 위한 subscript operator를 정의한다. C나 C++ 언어에서는 0이 첫 번째 원소를 가리키는 subscript operator를 사용하지만, **point** 객체에서는  $i$  번째 원소는 인덱스  $i$ 가 가리키도록 구현한다. 특히 subscript operator는 **const** 객체와 non-**const** 객체를 대상으로 호출하는 method를 각각 정의하는데, 코드 중복을 피하기 위하여 후자는 전자에 type casting을 활용하여 정의한다.

비상수 객체를 대상으로 하는 **operator()**가 상수 버전의 **operator()**를 호출하도록 하기 위하여, 비상수 **operator()** 안에서 단순히 **operator()**를 다시 호출하면 그 자신이 재귀적으로 호출된다. 즉 무한 재귀 호출이 되는데, 이것을 방지하기 위하여 “상수 버전의 **operator()**를 호출하고 싶다”는 의미를 코드에 표현해야 한다. 이 때 직접적인 방법이 없으므로 **\*this**를 타입 캐스팅해서 비상수 버전의 객체를 상수버전의 객체로 바꾼다. 이는 안전한 타입 변환을 강제로 수행하는 것이므로, **static\_cast**만 사용해도 충분하다. 반면, 상수 버전의 **operator()**를 호출해서 반환 받은 객체에서 상수성을 제거하고 비상수 객체를 반환해야 하므로, **const**를 제거해야 하는데, 이는 **const\_cast** 이외의 다른 방법이 없다. 따라서, 비상수 버전의 **operator()**는 다음의 순서대로 작동한다.

1. (**\*this**)의 타입에 **static\_cast**를 적용하여 **const** 객체로 변환.
2. 상수 버전의 **operator()**를 호출.
3. 돌려 받은 **double &** 타입에 **const\_cast**를 적용하여 상수성을 제거.

끝으로, C나 C++의 일반적인 컨벤션과 달리, 이 연산자는 첫 번째 원소를 얻기 위하여 1을 입력 인자로 넘겨줘야 한다. 주어진 인자가 **point** 객체의 차원을 벗어나면 첫 번째 좌표를 반환한다.

〈Operators of **point** 14〉 +≡

```

203 const double &point::operator()(const size_t &i) const {
204     size_t size = _elem.size();
205     if ((i < 1) ∨ (size < i)) {
206         return _elem[0];
207     } else {
208         return _elem[i - 1];
209     }
210 }
211 double &point::operator()(const size_t &i) {
212     return const_cast<double &>(static_cast<const point &>(*this)(i));
213 }

```

19.  $\langle$  Methods of **point 11**  $\rangle + \equiv$ 

```

214     const double &operator()(const size_t &) const;
215     double &operator()(const size_t &);

```

20. *dist()* method는 본 객체와 다른 **point** 타입 객체 사이의 거리(Euclidean distance, 2-norm)을 계산한다. 편의상, 두 객체가 같은 차원의 공간에 놓인 점들이 아니라면 -1.0을 반환한다.

 $\langle$  Other member functions of **point 20**  $\rangle \equiv$ 

```

216     double point::dist(const point &pt) const {
217         if (this->dim() != pt.dim()) return -1.;
218         size_t n = this->dim();
219         double sum = 0.0;
220         for (size_t i = 0; i != n; i++) {
221             sum += (_elem[i] - pt._elem[i]) * (_elem[i] - pt._elem[i]);
222         }
223         return std::sqrt(sum);
224     }

```

22번 마디도 살펴보라.

이 코드는 9번 마디에서 사용된다.

21.  $\langle$  Methods of **point 11**  $\rangle + \equiv$ 

```

225     double dist(const point &) const;

```

22. Debugging을 위해 **point** 타입 객체에 대한 정보를 출력하는 method를 정의한다.

 $\langle$  Other member functions of **point 20**  $\rangle + \equiv$ 

```

226     string point::description() const {
227         stringstream buffer;
228         buffer << "(";
229         for (size_t i = 0; i != dim() - 1; i++) {
230             buffer << _elem[i] << ", ";
231         }
232         buffer << _elem[dim() - 1] << ")" << endl;
233         return buffer.str();
234     }

```

23.  $\langle$  Methods of **point 11**  $\rangle + \equiv$ 

```

235     string description() const;

```

**24.** `point` 타입의 member method는 아니지만 두 `point` 객체 사이의 거리를 계산하기 위한 utility 함수를 정의한다.

〈Non-member functions for `point` 16〉 +=

```
236     double cagd::dist(const point &pt1, const point &pt2) {
237         return pt1.dist(pt2);
238     }
```

**25.** 〈Declaration of `cagd` functions 5〉 +=

```
239     double dist(const point &, const point &);
```

**26.** Test of `point` type. `point` 객체의 생성과 간단한 연산 기능들을 테스트하고 사용예시를 보여준다.

〈Test routines 26〉 ≡

```
240     print_title("operations_on_point_type");
241     {
242         point p0(3);
243         cout << "Dimension_of_p0=" << p0.dim() << ": ";
244         for (size_t i = 0; i < p0.dim(); i++) {
245             cout << p0(i) << " ";
246         }
247         cout << "\n\n";
248         point p1({1., 2., 3.});
249         cout << "Dimension_of_p1=" << p1.dim() << ": ";
250         for (size_t i = 0; i < p1.dim(); i++) {
251             cout << p1(i) << " ";
252         }
253         cout << "\n\n";
254         point p2({2., 4., 6.});
255         point p3 = .5 * p1 + .5 * p2;
256         cout << "p3=.5(1,2,3)+.5(2,4,6)=";
257         for (size_t i = 0; i < p3.dim(); i++) {
258             cout << p3(i) << " ";
259         }
260         cout << "\n\n";
261         cout << "Distance_from_p0_to_p1=" << dist(p0, p1) << "\n";
262         cout << "(It_should_be_3.741657387)\n\n";
263     }
```

90, 132, 142, 159, 164, 166번 마디도 살펴보자.

이 코드는 6번 마디에서 사용된다.

**27. Generic Curve.** `curve` 타입은 컨트롤 포인트에 의하여 모양과 특성이 결정되는 일반적인 곡선을 의미한다. 따라서 데이터 멤버로 `_ctrl_pts`를 갖는다.

한편, 그래픽스 객체를 다루는 경우 가장 쉽고 직관적인 디버깅 방법은 객체를 시각화하는 것이다. `curve` 타입은 PostScript 파일 출력을 위한 data member와 method들을 정의한다.

〈Definition of `curve` 27〉 ≡

```

264     class curve {
265     protected:
266         vector<point> _ctrl_pts;
267         mutable cagd::err_code _err;
268     public:
269         typedef vector<point>::iterator ctrlpt_itr;
270         typedef vector<point>::const_iterator const_ctrlpt_itr;
271         〈Methods of curve 30〉
272     };

```

이 코드는 2번 마디에서 사용된다.

**28.** `curve` 타입에는 PostScript 파일 출력을 위한 method들과 주어진 parameter에 대응하는 곡선의 값과 미분값을 계산하기 위한 method들의 인터페이스를 정의한다. 인터페이스는 pure virtual function으로 정의하므로 구현은 없다.

〈Implementation of `curve` 28〉 ≡

```

273     〈Properties of curve 29〉
274     〈Access control points of curve 31〉
275     〈Constructor and destructor of curve 36〉
276     〈Methods for debugging of curve 38〉
277     〈Operators of curve 40〉

```

이 코드는 3번 마디에서 사용된다.

**29.** `curve` 타입 객체의 property들 중, 차원을 반환하는 method는 컨트롤 포인트의 차원에 의하여 결정된다. 하지만 곡선의 차수는 아직 정의할 수 없으므로 pure virtual function으로 둔다.

〈Properties of `curve` 29〉 ≡

```

278     unsigned long curve::dimension() const {
279         if (_ctrl_pts.size() > 0) {
280             return _ctrl_pts.begin()-dim();
281         } else {
282             return 0;
283         }
284     }
285     unsigned long curve::dim() const {
286         return dimension();
287     }

```

이 코드는 28번 마디에서 사용된다.

30. 〈Methods of **curve 30**〉 ≡

```

288 public:
289     virtual unsigned long dimension() const;
290     virtual unsigned long dim() const;
291     virtual unsigned long degree() const = 0;

```

32, 33, 35, 37, 39, 41번 마디도 살펴보라.

이 코드는 27번 마디에서 사용된다.

31. **curve**의 컨트롤 포인트에 접근하기 위한 method를 정의한다. 이 method는 인자 0이 주어졌을 때, 첫 번째 컨트롤 포인트를 반환한다.

〈Access control points of **curve 31**〉 ≡

```

292     point curve::ctrl_pts(const size_t &i) const {
293         size_t size = _ctrl_pts.size();
294         if ((i < 1) ∨ (size < i)) {
295             return _ctrl_pts[0];
296         } else {
297             return _ctrl_pts[i];
298         }
299     }
300     size_t curve::ctrl_pts_size() const {
301         return _ctrl_pts.size();
302     }

```

이 코드는 28번 마디에서 사용된다.

32. 〈Methods of **curve 30**〉 +≡

```

303     point ctrl_pts(const size_t &) const;
304     size_t ctrl_pts_size() const;

```

**33.** 곡선, control polygon, control point 들을 PostScript 파일로 출력하는 함수들의 인터페이스는 pure virtual function으로 정의한다.

〈Methods of **curve 30**〉 +≡

```

305     virtual void write_curve_in_postscript(
306         psf &,
307         unsigned, float,
308         int x = 1, int y = 1,
309         float magnification = 1.) const = 0;
310     virtual void write_control_polygon_in_postscript(
311         psf &,
312         float,
313         int x = 1, int y = 1,
314         float magnification = 1.) const = 0;
315     virtual void write_control_points_in_postscript(
316         psf &,
317         float,
318         int x = 1, int y = 1,
319         float magnification = 1.) const = 0;

```

**34.** 〈Error codes of **cagd 34**〉 ≡

```

320     OUTPUT_FILE_OPEN_FAIL ,

```

51, 66, 71, 121, 149, 156, 169, 177번 마디도 살펴보라.  
이 코드는 2번 마디에서 사용된다.

**35.** 곡선 위에 있는 점의 위치와 미분을 계산하는 함수들도 pure virtual function으로 정의한다.

〈Methods of **curve 30**〉 +≡

```

321     public:
322         virtual point evaluate(const double) const = 0;
323         virtual point derivative(const double) const = 0;

```



**36.** Constructor와 destructor에 특별한 것은 없다.

⟨Constructor and destructor of **curve 36**⟩ ≡

```

324     curve::curve() {}
325     curve::curve(const vector<point> &pts)
326         : _ctrl_pts(pts) {}
327     curve::curve(const list<point> &pts)
328         : _ctrl_pts(vector<point>(pts.size(), pts.begin()-dim())) {}
329     list<point>::const_iterator pt(pts.begin());
330     for (size_t i = 0; i ≠ pts.size(); i++) {
331         _ctrl_pts[i] = *pt;
332         pt++;
333     }
334 }
335     curve::curve(const curve &src)
336         : _ctrl_pts(src._ctrl_pts) {}
337     curve::~~curve() {}

```

이 코드는 28번 마디에서 사용된다.

**37.** ⟨Methods of **curve 30**⟩ +=

```

338     public:
339     curve();
340     curve(const vector<point> &);
341     curve(const list<point> &);
342     curve(const curve &);
343     virtual ~curve();

```

**38.** Debugging을 위해 **curve** 타입 객체의 정보를 출력하는 method를 정의한다.

⟨Methods for debugging of **curve 38**⟩ ≡

```

344     string curve::description() const {
345         stringstream buffer;
346         buffer << "-----" << endl;
347         buffer << "Description of Curve" << endl;
348         buffer << "-----" << endl;
349         buffer << "Dimension of curve:" << dim() << endl;
350         buffer << "Control points:" << endl;
351         for (size_t i = 0; i ≠ _ctrl_pts.size(); i++) {
352             buffer << "    " << _ctrl_pts[i].description();
353         }
354         return buffer.str();
355     }

```

이 코드는 28번 마디에서 사용된다.

**39.**  $\langle \text{Methods of curve 30} \rangle + \equiv$

```
356 public:
357     string description() const;
```

**40.** Assignment operator.

$\langle \text{Operators of curve 40} \rangle \equiv$

```
358     curve &curve::operator=(const curve &crv) {
359         _ctrl_pts = crv._ctrl_pts;
360         _err = crv._err;
361         return *this;
362     }
```

이 코드는 28번 마디에서 사용된다.

**41.**  $\langle \text{Methods of curve 30} \rangle + \equiv$

```
363 public:
364     curve &operator=(const curve &);
```

**42. Bézier Curve.**

**bezier** 타입은  $n$ -차원 유클리드 공간에 존재하는 컨트롤 포인트를 갖는 Bézier 곡선을 기술한다. 앞에서 정의한 **curve** 타입의 파생 클래스 (derived class)로 정의한다.

〈Definition of **bezier** 42〉 ≡

```

365     class bezier : public curve {
366         〈Data members of bezier 43〉
367         〈Methods of bezier 46〉
368     };

```

이 코드는 2번 마디에서 사용된다.

**43. bezier** 타입은 Bézier 곡선의 차수를 저장하기 위한 *\_degree* 변수와, 실제 컨트롤 포인트들을 저장하고 위한 *\_ctrl\_pts* 변수는 **curve** 타입에서 상속받는다.

〈Data members of **bezier** 43〉 ≡

```

369     protected:
370         unsigned long _degree;

```

이 코드는 42번 마디에서 사용된다.

**44. bezier** 타입에 대한 method들은 다음과 같다.

1. Properties;
2. Constructor들과 destructor;
3. Operators;
4. 곡선상 점들의 위치와 속도, 곡률을 계산하는 methods;
5. 곡선을 임의의 점에서 분할하는 method;
5. 곡선의 차수를 높이거나 낮추기 위한 methods;
6. PostScript 파일로 출력하기 위한 methods.

〈Implementation of **bezier** 44〉 ≡

```

371     〈Properties of bezier 45〉
372     〈Constructors and destructor of bezier 47〉
373     〈Operators of bezier 49〉
374     〈Evaluation of bezier 52〉
375     〈Subdivision of bezier 58〉
376     〈Degree elevation and reduction of bezier 64〉
377     〈Output to PostScript of bezier 72〉

```

이 코드는 3번 마디에서 사용된다.

**45.** **bezier** 타입의 대표적인 property는 곡선의 차수(degree)와 차원(dimension)이다. 차원에 대한 것은 **curve** 타입에서 정의했으므로, **bezier** 타입에서 별도로 정의하지는 않는다.

〈Properties of **bezier** 45〉 ≡

```
378     unsigned long bezier::degree() const {
379         return _degree;
380     }
```

이 코드는 44번 마디에서 사용된다.

**46.** 〈Methods of **bezier** 46〉 ≡

```
381     public:
382     unsigned long degree() const;
```

48, 50, 53, 55, 63, 65, 70, 73번 마디도 살펴보라.

이 코드는 42번 마디에서 사용된다.

**47.** 몇 가지 생성자들을 정의한다. 간단한 복사 생성자와 standard library의 **vector** 또는 **list**를 이용하여 컨트롤 포인트들을 넘겨 받았을 때 곡선을 생성하는 생성자들이다.

〈Constructors and destructor of **bezier** 47〉 ≡

```
383     bezier::bezier() {}
384     bezier::bezier(const bezier &src) {
385         _degree = src._degree;
386         _ctrl_pts.clear();
387         for (size_t i = 0; i ≠ src._ctrl_pts.size(); ++i) {
388             _ctrl_pts.push_back(src._ctrl_pts[i]);
389         }
390     }
391     bezier::bezier(vector<point> points) {
392         _degree = points.size() - 1;
393         _ctrl_pts.clear();
394         for (size_t i = 0; i ≠ points.size(); ++i) {
395             _ctrl_pts.push_back(points[i]);
396         }
397     }
398     bezier::bezier(list<point> points) {
399         _degree = points.size() - 1;
400         _ctrl_pts.clear();
401         for (list<point>::const_iterator i = points.begin(); i ≠ points.end(); i++) {
402             _ctrl_pts.push_back(*i);
403         }
404     }
405     bezier::~bezier() {}
```

이 코드는 44번 마디에서 사용된다.

48. 〈Methods of **bezier** 46〉 +≡

```

406 public:
407     bezier();
408     bezier(const bezier &);
409     bezier(vector<point>);
410     bezier(list<point>);
411     virtual ~bezier();

```

49. 다른 **bezier** 객체로부터의 assignment operator.〈Operators of **bezier** 49〉 ≡

```

412     bezier &bezier::operator=(const bezier &src) {
413         _degree = src._degree;
414         curve::operator=(src);
415         return *this;
416     }

```

이 코드는 44번 마디에서 사용된다.

50. 〈Methods of **bezier** 46〉 +≡

```

417 public:
418     bezier &operator=(const bezier &);

```

51. 〈Error codes of **cagd** 34〉 +≡

```

419     DEGREE_MISMATCH ,

```

52. Bézier 곡선상 각 점의 위치와 속도는 de Casteljau의 recursive linear interpolation algorithm을 이용한다.

〈Evaluation of **bezier** 52〉 ≡

```

420     point bezier::evaluate(const double t) const {
421         vector<point> coeff;
422         for (size_t i = 0; i < _ctrl_pts.size(); ++i) {
423             coeff.push_back(_ctrl_pts[i]);
424         }
425         double t1 = 1.0 - t;
426         for (size_t r = 1; r < _degree + 1; r++) {
427             for (size_t i = 0; i < _degree - r + 1; i++) {
428                 coeff[i] = t1 * coeff[i] + t * coeff[i + 1];
429             }
430         }
431         return coeff[0];
432     }
433     point bezier::derivative(const double t) const {
434         vector<point> coeff;
435         for (size_t i = 0; i < _ctrl_pts.size() - 1; ++i) {
436             coeff.push_back(_degree * (_ctrl_pts[i + 1] - _ctrl_pts[i]));
437         }
438         double t1 = 1.0 - t;
439         for (size_t r = 1; r < _degree; r++) {
440             for (size_t i = 0; i < _degree - r; i++) {
441                 coeff[i] = t1 * coeff[i] + t * coeff[i + 1];
442             }
443         }
444         return coeff[0];
445     }

```

54번 마디도 살펴보라.

이 코드는 44번 마디에서 사용된다.

53. 〈Methods of **bezier** 46〉 + ≡

```

446     public:
447         point evaluate(const double) const;
448         point derivative(const double) const;

```

**54.** Bézier 곡선의 임의의 점에서 곡률을 계산하는 method를 정의한다. *curvature\_at\_zero()* 함수는 곡선 시작점에서의 곡률을 계산한다. *signed\_curvature()* 함수는 *b*부터 *e*까지로 한정되는 곡선의 일부 구간에 대하여 곡률을 계산한다. 먼저 곡률을 계산할 구간을 *density*개의 등간격으로 나누고, 각 지점에서 Bézier 곡선의 subdivision을 구한다. 계산의 수치적 안정성을 위하여 둘로 나뉜 곡선 조각들 중 큰 쪽에서 *curvature\_at\_zero()* 함수를 이용하여 곡률을 계산하고 그 결과를 하나의 **vector** 객체에 담아 반환한다. *curvature\_at\_zero()* 함수는 *signed\_area()* 함수를 이용하여 부호가 붙은 곡률을 반환하므로, 곡선의 전반부에서 계산하는 곡률은 부호를 반대로 뒤집어서 반환함에 유의한다.

〈Evaluation of **bezier** 52〉 +=

```

449     double
450     bezier::curvature_at_zero() const {
451         double dist = cagd::dist(_ctrl_pts[0], _ctrl_pts[1]);
452         return 2.0 * (_degree - 1) *
453         cagd::signed_area(_ctrl_pts[0], _ctrl_pts[1], _ctrl_pts[2]) / (_degree * dist * dist * dist);
454     }
455     vector<point>
456     bezier::signed_curvature(const unsigned density, const double b, const double e) const {
457         /* b: begin of the interval. e: end of the interval. */
458         double delta = (e - b) / density;
459         unsigned half = density / 2;
460         vector<point> kappa;
461         for (size_t i = 0; i <= density; i++) {
462             double t = b + i * delta;
463             bezier left(*this);
464             bezier right(*this);
465             if (i <= half) {
466                 subdivision(t, left, right);
467                 double h = right.curvature_at_zero();
468                 kappa.push_back(point({t, h}));
469             } else {
470                 subdivision(t, left, right);
471                 double h = left.curvature_at_zero();
472                 kappa.push_back(point({t, std::fabs(-h)}));
473             }
474         }
475         return kappa;
476     }

```

55. 〈Methods of **bezier** 46〉 +≡

```

476 public:
477     double curvature_at_zero() const;
478     vector<point> signed_curvature(const unsigned, const double b = 0., const double e = 1.) const;

```

56. *signed\_area()* 함수는 2-차원 평면상에 존재하는 세 개의 점으로 이루어지는 삼각형의 면적을 계산한다.

〈Implementation of **cagd** functions 4〉 +≡

```

479     double cagd::signed_area(const point p1, const point p2, const point p3) {
480         double area;
481         area = ((p2(1) - p1(1)) * (p3(2) - p1(2)) - (p2(2) - p1(2)) * (p3(1) - p1(1))) / 2.0;
482         return area;
483     }

```

57. 〈Declaration of **cagd** functions 5〉 +≡

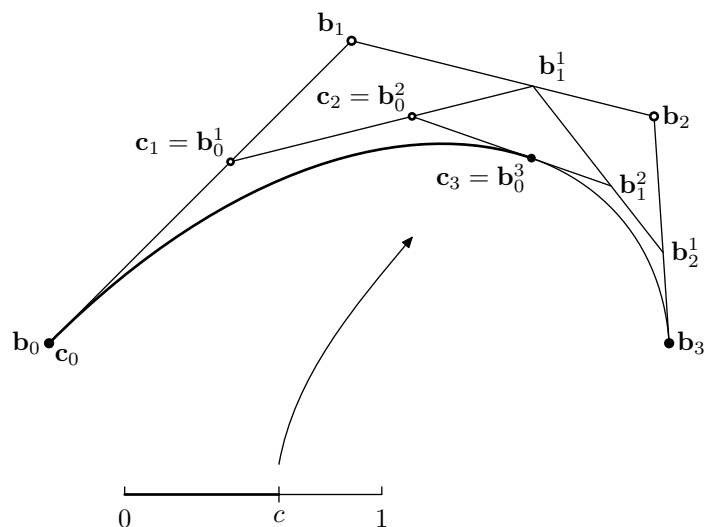
```

484     double signed_area(const point, const point, const point);

```



58. Bézier 곡선을 임의의 점에서 두 개의 곡선으로 분할하는 method를 정의한다. 이해를 돕기 위해 컨트롤 포인트  $\mathbf{b}_0, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3$ 로 정의되는 3차 Bézier 곡선을 파라미터  $c$ 인 지점에서 둘로 나누는 과정을 설명한다. Bézier 곡선을 두개로 분할하고 새로운 컨트롤 포인트들을 구하는 것은 de Casteljau 알고리즘을 적용하는 과정과 동일하다. 즉, 선분  $\mathbf{b}_i\text{--}\mathbf{b}_{i+1}$ 을  $c : 1 - c$ 로 내분하는 점을  $\mathbf{b}_i^1(c)$ , ( $i = 0, 1, 2$ )이라 하고, 다시 선분  $\mathbf{b}_i^1(c)\text{--}\mathbf{b}_{i+1}^1(c)$ 를  $c : 1 - c$ 로 내분하는 점을  $\mathbf{b}_i^2(c)$  ( $i = 0, 1$ ), 또 선분  $\mathbf{b}_i^2(c)\text{--}\mathbf{b}_{i+1}^2(c)$ 를  $c : 1 - c$ 로 내분하는 점을  $\mathbf{b}_i^3(c)$  ( $i = 0$ )이라 하자. 그러면 파라미터  $[0, c]$  구간에 해당하는 곡선의 분할에 대한 새로운 컨트롤 포인트들은  $\mathbf{c}_0 = \mathbf{b}_0$ ,  $\mathbf{c}_1 = \mathbf{b}_0^1(c)$ ,  $\mathbf{c}_2 = \mathbf{b}_0^2(c)$ ,  $\mathbf{c}_3 = \mathbf{b}_0^3(c)$ 가 된다.  $[c, 1]$  구간도 마찬가지로 de Casteljau 알고리즘에 의해 얻어지는 중간 단계의 점들이 새로운 컨트롤 포인트가 된다.



〈Subdivision of **bezier** 58〉 ≡

```

485 void bezier :: subdivision(double t, bezier &left, bezier &right) const {
486     double t1 = 1.0 - t;
487     vector<point> points;    /* temporary store */
488     〈Obtain the right subpolygon of Bézier curve 59〉;
489     〈Obtain the left subpolygon of Bézier curve 61〉;
490 }

```

이 코드는 44번 마디에서 사용된다.

**59.** 우측, 즉 파라미터  $[c, 1]$  구간에 대한 control polygon을 구한다. 먼저 control point들을 temporary store에 복사하고, 그 point들에 de Casteljau 알고리즘을 적용하여 subpolygon의 control point들을 구한다. Temporary store들어 있던 결과가 우측 부분 곡선의 control point들이므로 그것들을 복사해 온다.

〈Obtain the right subpolygon of Bézier curve 59〉≡

```

491     right._ctrl_pts.clear();
492     right._degree = _degree;
493     for (size_t i = 0; i ≠ _ctrl_pts.size(); i++) {
494         points.push_back(_ctrl_pts[i]);
495     }
496     〈Obtain the right subpolygon using the de Casteljau algorithm 60〉;
497     for (size_t i = 0; i ≠ (_degree + 1); i++) {
498         right._ctrl_pts.push_back(points[i]);
499     }
```

이 코드는 58번 마디에서 사용된다.

**60.** 〈Obtain the right subpolygon using the de Casteljau algorithm 60〉≡

```

500     for (size_t r = 1; r ≠ _degree + 1; r++) {
501         for (size_t i = 0; i ≠ _degree - r + 1; i++) {
502             points[i] = t1 * points[i] + t * points[i + 1];
503         }
504     }
```

이 코드는 59번 마디에서 사용된다.

**61.** 왼쪽, 즉 파라미터  $[0, c]$  구간에 대한 control polygon을 구한다. 방법은 오른쪽 부분 곡선을 구할때와 마찬가지로인데, control point들을 temporary store에 역순으로 복사하고  $t$ 를  $1 - t$ 로 바꿔 놓은 후 de Casteljau 알고리즘을 적용한다. 즉, 곡선과 파라미터를 모두 뒤집어 놓고 같은 과정을 반복하는 것이다.

〈Obtain the left subpolygon of Bézier curve 61〉≡

```

505     t = 1.0 - t;
506     t1 = 1.0 - t1;
507     points.clear();
508     left._ctrl_pts.clear();
509     left._degree = _degree;
510     unsigned long index = _degree;
511     for (size_t i = 0; i ≠ _ctrl_pts.size(); i++) {      /* Reverse order. */
512         points[index--] = _ctrl_pts[i];
513     }
514     〈Obtain the left subpolygon using de Casteljau algorithm 62〉;
515     for (size_t i = 0; i ≠ _degree + 1; i++) {
516         left._ctrl_pts.push_back(points[i]);
517     }
```

이 코드는 58번 마디에서 사용된다.

**62.** 〈Obtain the left subpolygon using de Casteljau algorithm 62〉 ≡

```

518   for (size_t r = 1; r < _degree + 1; r++) {
519       for (size_t i = 0; i < _degree - r + 1; i++) {
520           points[i] = t1 * points[i] + t * points[i + 1];
521       }
522   }
```

이 코드는 61번 마디에서 사용된다.

**63.** 〈Methods of **bezier** 46〉 +≡

```

523   public:
524       void subdivision(const double, bezier &, bezier &) const;
```

**64.** Bézier 곡선의 차수를 높이는 method를 구현한다. *elevate\_degree()*는 Bézier 곡선의 차수를 하나 높이며, 여러 차수를 한번에 높이려면 recursion을 수행한다. 따라서 method 시작부분에서는 오류처리와 종료조건을 점검하며, 그 이후에는 컨트롤 포인트를 하나 추가하는 작업을 한다. 만약 현재 곡선의 차수보다 낮은 차수로 올리려고 하면 (nonsense!), 객체 내에 DEGREE\_ELEVATION\_FAIL 오류코드를 저장하고 바로 반환한다.

〈Degree elevation and reduction of **bezier** 64〉 ≡

```

525   void bezier::elevate_degree(unsigned long dgr) {
526       if (_degree > dgr) {
527           _err = DEGREE_ELEVATION_FAIL;
528           return;
529       }
530       if (_degree == dgr) {
531           return;
532       }
533       _degree++;
534       point backup_point = _ctrl_pts[0];
535       unsigned long counter = 1;
536       for (size_t i = 1; i < _ctrl_pts.size(); ++i) {
537           point tmp_point = backup_point;
538           backup_point = _ctrl_pts[i];
539           double ratio = double(counter)/double(_degree);
540           _ctrl_pts[i] = ratio * tmp_point + (1.0 - ratio) * backup_point;
541           counter++;
542       }
543       _ctrl_pts.push_back(backup_point);
544       return elevate_degree(dgr);
545   }
```

69번 마디도 살펴보라.

이 코드는 44번 마디에서 사용된다.

65. 〈Methods of **bezier** 46〉 +≡

546 **public:**

547 **void** *elevate\_degree*(**unsigned long**);

66. 〈Error codes of **cagd** 34〉 +≡

548 **DEGREE\_ELEVATION\_FAIL** ,

67. 다음에 정의할 함수를 위해 먼저 *factorial*을 구하는 함수를 **cagd** namespace에 정의한다.

〈Implementation of **cagd** functions 4〉 +≡

549 **unsigned long cagd::factorial**(**unsigned long** *n*) {

550 **if** (*n* ≤ 0) {

551 **return** 1<sub>UL</sub>;

552 } **else** {

553 **return** *n* \* *factorial*(*n* − 1);

554 }

555 }

68. 〈Declaration of **cagd** functions 5〉 +≡

556 **unsigned long factorial**(**unsigned long**);

**69.** Bézier 곡선의 차수를 낮추는 method를 구현한다. 이 함수도 차수를 하나씩 낮추도록 구현되어 있으며, 한번에 여러 차수를 낮추려면 recursion을 수행한다.

앞에서 설명했듯이,  $n$ 차 Bézier 곡선을 정확하게  $n+1$ 차 Bézier 곡선으로 차수를 높이는 것은 가능하지만,  $n+1$ 차 Bézier 곡선의 형상 변화 없이  $n$ 차 Bézier 곡선으로 차수를 낮추는 것은 불가능하다. 어느 정도 곡선의 변화를 수반할 수 밖에 없는데, 이는  $n+2$ 개의 컨트롤 포인트들,  $\mathbf{b}_i^{(1)} (i = 0, \dots, n+1)$ 을  $n+1$ 개의 컨트롤 포인트들,  $\mathbf{b}_i (i = 0, \dots, n)$ 로 근사화하는 다음의 문제로 이해할 수 있다. ( $n$ 차 Bézier 곡선은  $n+1$ 개의 컨트롤 포인트들을 갖는다.)

$$\begin{pmatrix} 1 & & & & & \\ * & * & & & & \\ & * & * & & & \\ & & & \ddots & & \\ & & & & * & * \\ & & & & & 1 \end{pmatrix} \begin{pmatrix} \mathbf{b}_0 \\ \vdots \\ \mathbf{b}_n \end{pmatrix} = \begin{pmatrix} \mathbf{b}_0^{(1)} \\ \vdots \\ \mathbf{b}_{n+1}^{(1)} \end{pmatrix}.$$

이를 다시 줄여 쓰면,

$$M\mathbf{B} = \mathbf{B}^{(1)}$$

이며,  $M$ 은  $(n+2) \times (n+1)$  행렬이다. 이는 정방행렬이 아니므로 위의 등식을 풀기 위하여 양변에  $M^\top$ 을 곱하면,

$$M^\top M\mathbf{B} = M^\top \mathbf{B}^{(1)}$$

으로  $M^\top M$ 이 정방행렬이므로 역행렬을 구해서 양변에 곱함으로써 해를 구할 수 있다.  $M$  행렬 주대각의 첫 번째원소와 마지막 원소가 1인 것은 Bézier 곡선의 차수를 낮추더라도 시작점과 끝점은 그대로 유지하기 위함이다.

만약 현재 곡선의 차수보다 높은 차수로 낮추려고 하면 (nonsense!) DEGREE\_REDUCTION\_FAIL 오류코드를 남기고 method는 즉시 반환한다.

⟨Degree elevation and reduction of **bezier** 64⟩ +≡

```

557 void bezier::reduce_degree(const unsigned long dgr) {
558     if (_degree < dgr) {
559         _err = DEGREE_REDUCTION_FAIL;
560         return;
561     }
562     if (_degree == dgr) {
563         return;
564     }
565     vector<point> l2r;
566     l2r.push_back(_ctrl_pts[0]);
567     unsigned long counter = 1;
568     for (size_t i = 1; i != _ctrl_pts.size() - 1; ++i) {
569         l2r.push_back(((double)(_degree) * _ctrl_pts[i] - double(counter) * (l2r.back())) / double(_degree -
                    counter));
570         counter++;
571     }
572     vector<point> r2l_reversed;
```

```

573     r2l_reversed.push_back(_ctrl_pts.back());
574     counter = _degree;
575     for (size_t i = _ctrl_pts.size() - 2; i >= 0; --i) {
576         r2l_reversed.push_back(((double(_degree) * (_ctrl_pts[i]) - double(_degree - counter) *
577             r2l_reversed.front()) / double(counter));
578         counter--;
579     }
580     vector<point> r2l;
581     size_t r2l_reversed_size = r2l_reversed.size();
582     for (size_t i = 0; i < r2l_reversed_size; i++) {
583         r2l.push_back(r2l_reversed.back());
584         r2l_reversed.pop_back();
585     }
586     point backup1 = _ctrl_pts[0];
587     point backup2 = _ctrl_pts.back();
588     _ctrl_pts.clear();
589     _ctrl_pts.push_back(backup1);
590     for (size_t i = 1; i <= _degree - 2; ++i) {
591         unsigned long combi = 0;
592         for (size_t j = 0; j <= i; ++j) {
593             combi += cagd::factorial(2 * _degree) / (cagd::factorial(2 * j) * cagd::factorial(2 * (_degree - j)));
594         }
595         double lambda = double(combi) / std::pow(2., 2 * _degree - 1);
596         _ctrl_pts.push_back((1.0 - lambda) * l2r[i] + lambda * r2l[i]);
597     }
598     _ctrl_pts.push_back(backup2);
599     _degree--;
600     return reduce_degree(dgr);

```

70. <Methods of **bezier** 46> +≡

```

601 public:
602     void reduce_degree(const unsigned long);

```

71. <Error codes of **cagd** 34> +≡

```

603     DEGREE_REDUCTION_FAIL ,

```

**72.** Bézier curve의 PostScript 출력을 위한 몇 가지 함수들을 정의한다. `write_curve_in_postscript()` 함수는 Bézier 곡선을 그리기 위한 함수. PostScript은 2-차원 평면 용지에 페이지를 기술하는 언어이므로,  $n$ -차원 공간에 존재하는 Bézier 곡선의 몇 번째와 몇 번째 좌표를 그릴 것인지 지정해야 한다. 만약 아무런 지정이 없으면, 첫 번째와 두 번째 좌표를 출력한다.

〈Output to PostScript of **bezier** 72〉≡

```

604 void bezier::write_curve_in_postscript(
605     psf &ps_file,
606     unsigned step,
607     float line_width,
608     int x, int y,
609     float magnification
610 ) const {
611     ios_base::fmtflags previous_options = ps_file.flags();
612     ps_file.precision(4);
613     ps_file.setf(ios_base::fixed, ios_base::floatfield);
614     ps_file << "newpath" << endl << "[ ] 0 setdash" << line_width << " setlinewidth" << endl;
615     point pt = magnification * evaluate(0);
616     ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
617     for (size_t i = 1; i ≤ step; i++) {
618         double t = double(i)/double(step);
619         pt = magnification * evaluate(t);
620         ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
621     }
622     ps_file << "stroke" << endl;
623     ps_file.flags(previous_options);
624 }
625 void bezier::write_control_polygon_in_postscript(
626     psf &ps_file,
627     float line_width,
628     int x, int y,
629     float magnification
630 ) const {
631     ios_base::fmtflags previous_options = ps_file.flags();
632     ps_file.precision(4);
633     ps_file.setf(ios_base::fixed, ios_base::floatfield);
634     ps_file << "newpath" << endl;
635     ps_file << "[ ] 0 setdash" << .5 * line_width << " setlinewidth" << endl;
636     point pt = magnification * _ctrl_pts[0];
637     ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
638     for (size_t i = 1; i ≠ _ctrl_pts.size(); ++i) {

```

```

639     pt = magnification * _ctrl_pts[i];
640     ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
641 }
642 ps_file << "stroke" << endl;
643 ps_file.flags(previous_options);
644 }
645 void bezier::write_control_points_in_postscript(
646     psf &ps_file,
647     float line_width,
648     int x, int y,
649     float magnification
650 ) const {
651     ios_base::fmtflags previous_options = ps_file.flags();
652     ps_file.precision(4);
653     ps_file.setf(ios_base::fixed, ios_base::floatfield);
654     ps_file << "0 setgray" << endl;
655     ps_file << "newpath" << endl;
656     point pt = magnification * _ctrl_pts[0];
657     ps_file << pt(x) << "\t" << pt(y) << "\t";
658     ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
659     ps_file << "closepath" << endl;
660     ps_file << "fill stroke" << endl;
661     if (_ctrl_pts.size() > 2) {
662         for (size_t i = 1; i < _ctrl_pts.size() - 1; ++i) {
663             ps_file << "newpath" << endl;
664             pt = magnification * _ctrl_pts[i];
665             ps_file << pt(x) << "\t" << pt(y) << "\t";
666             ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
667             ps_file << "closepath" << endl;
668             ps_file << line_width << "\t" << "setlinewidth" << endl;
669             ps_file << "stroke" << endl;
670         }
671         ps_file << "0 setgray" << endl;
672         ps_file << "newpath" << endl;
673         pt = magnification * _ctrl_pts.back();
674         ps_file << pt(x) << "\t" << pt(y) << "\t";
675         ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
676         ps_file << "closepath" << endl;
677         ps_file << "fill stroke" << endl;
678     }
679     ps_file.flags(previous_options);

```



680 }

이 코드는 44번 마디에서 사용된다.

73. 〈Methods of **bezier** 46〉 +≡

```
681 void write_curve_in_postscript(  
682     psf &, unsigned, float, int x = 1, int y = 2,  
683     float magnification = 1.) const;  
684 void write_control_polygon_in_postscript(  
685     psf &, float, int x = 1, int y = 2,  
686     float magnification = 1.) const;  
687 void write_control_points_in_postscript(  
688     psf &, float, int x = 1, int y = 2,  
689     float magnification = 1.) const;
```

**74. Piecewise Bézier Curve.** 여러개의 Bézier curve들을 모아 한번에 다루기 위한 타입이다. **bezier** 타입 객체들을 저장하기 위한 **vector<bezier>** 타입의 데이터 멤버와 몇 가지 method들을 갖는다. 그리고 **vector**에 들어있는 객체들에 접근하기 위한 iterator의 타입을 선언한다.

〈Definition of **piecewise\_bezier\_curve** 74〉 ≡

```

690     class piecewise_bezier_curve : public curve {
691     protected:
692         vector<bezier> _curves;
693     public:
694         typedef vector<bezier>::const_iterator const_curve_itr;
695         typedef vector<bezier>::iterator curve_itr;
696         〈Methods of piecewise_bezier_curve 77〉
697     };

```

이 코드는 2번 마디에서 사용된다.

**75. piecewise\_bezier\_curve** 타입의 method들은 다음과 같다.

〈Implementation of **piecewise\_bezier\_curve** 75〉 ≡

```

698     〈Constructors and destructor of piecewise_bezier_curve 76〉
699     〈Properties of piecewise_bezier_curve 78〉
700     〈Modification of piecewise_bezier_curve 80〉
701     〈Operators of piecewise_bezier_curve 82〉
702     〈Degree elevation and reduction of piecewise_bezier_curve 84〉
703     〈Evaluation and derivative of piecewise_bezier_curve 86〉
704     〈PostScript output of piecewise_bezier_curve 88〉

```

이 코드는 3번 마디에서 사용된다.

**76. piecewise\_bezier\_curve** 타입은 default constructor와 copy constructor를 갖는다.

〈Constructors and destructor of **piecewise\_bezier\_curve** 76〉 ≡

```

705     piecewise_bezier_curve::piecewise_bezier_curve() {}
706     piecewise_bezier_curve::piecewise_bezier_curve(const piecewise_bezier_curve &r)
707     : curve::curve(r), _curves(r._curves) {}
708     piecewise_bezier_curve::~~piecewise_bezier_curve() {}

```

이 코드는 75번 마디에서 사용된다.

**77. 〈Methods of **piecewise\_bezier\_curve** 77〉 ≡**

```

709     public:
710         piecewise_bezier_curve();
711         piecewise_bezier_curve(const piecewise_bezier_curve &);
712         virtual ~piecewise_bezier_curve();

```

79, 81, 83, 85, 87, 89번 마디도 살펴보라.

이 코드는 74번 마디에서 사용된다.

**78.** `piecewise_bezier_curve` 타입 객체의 몇 가지 property들을 정의한다. 객체가 포함하는 Bézier 곡선이 모두 같은 차수를 갖는 것은 아니므로, `piecewise_bezier_curve` 타입 객체의 차수는 그것이 갖고 있는 Bézier 곡선들 중 가장 높은 차수로 정의한다. 그러나 차원은 모든 곡선들에 대하여 동일하므로, 편의상 첫 번째 곡선의 차원을 반환한다.

⟨Properties of `piecewise_bezier_curve` 78⟩ ≡

```

713     size_t piecewise_bezier_curve::count() const {
714         return _curves.size();
715     }
716     unsigned long piecewise_bezier_curve::dimension() const {
717         if (_curves.size() ≠ 0) {
718             return _curves.begin()→dimension();
719         } else {
720             return 0;
721         }
722     }
723     unsigned long piecewise_bezier_curve::dim() const {
724         return dimension();
725     }
726     unsigned long piecewise_bezier_curve::degree() const {
727         unsigned long dgr = 0;
728         for (const_curve_itr crv = _curves.begin(); crv ≠ _curves.end(); crv++) {
729             if (crv→degree() > dgr) {
730                 dgr = crv→degree();
731             }
732         }
733         return dgr;
734     }

```

이 코드는 75번 마디에서 사용된다.

**79.** ⟨Methods of `piecewise_bezier_curve` 77⟩ +≡

```

735     public:
736         size_t count() const;
737         unsigned long dimension() const;
738         unsigned long dim() const;
739         unsigned long degree() const;

```

**80.** `piecewise_bezier_curve` 타입 객체에 `bezier` 타입 객체를 추가하는 method를 정의한다.

〈Modification of `piecewise_bezier_curve` 80〉≡

```
740 void piecewise_bezier_curve::push_back(bezier crv) {
741     _curves.push_back(crv);
742 }
```

이 코드는 75번 마디에서 사용된다.

**81.** 〈Methods of `piecewise_bezier_curve` 77〉+≡

```
743 public:
744 void push_back(bezier);
```

**82.** Operators of `piecewise_bezier_curve`.

〈Operators of `piecewise_bezier_curve` 82〉≡

```
745 piecewise_bezier_curve &piecewise_bezier_curve::operator=(const piecewise_bezier_curve
    &crv) {
746     curve::operator=(crv);
747     _curves = crv._curves;
748     return *this;
749 }
```

이 코드는 75번 마디에서 사용된다.

**83.** 〈Methods of `piecewise_bezier_curve` 77〉+≡

```
750 public:
751 piecewise_bezier_curve &operator=(const piecewise_bezier_curve &);
```

**84.** `piecewise_bezier_curve` 타입 객체에 포함되어 있는 모든 곡선들의 차수를 높이거나 낮추는 method를 정의한다.

〈Degree elevation and reduction of `piecewise_bezier_curve` 84〉≡

```
752 void piecewise_bezier_curve::elevate_degree(const unsigned long dgr) {
753     for (curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
754         crv->elevate_degree(dgr);
755     }
756 }
757 void piecewise_bezier_curve::reduce_degree(const unsigned long dgr) {
758     for (curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
759         crv->reduce_degree(dgr);
760     }
761 }
```

이 코드는 75번 마디에서 사용된다.

85.  $\langle \text{Methods of } \texttt{piecewise\_bezier\_curve} \text{ 77} \rangle + \equiv$

```
762 public:
763     void elevate_degree(const unsigned long);
764     void reduce_degree(const unsigned long);
```

86. `piecewise_bezier_curve` 타입의 `evaluation`과 `derivative`를 구하는 `method`를 정의한다. 먼저 주어진 인자  $u$ 의 값을 보고 몇 번째 `bezier` 곡선에서 값을 구할지 결정한다. `piecewise_bezier_curve` 객체에 Bézier 곡선이  $n$ 개 포함되어 있다면,  $0 \leq u \leq n$ 이어야 한다. 만약 객체 내에 곡선이 하나도 없거나,  $u$ 가 적절한 범위 밖의 값으로 주어지면 0을 반환한다.

$\langle \text{Evaluation and derivative of } \texttt{piecewise\_bezier\_curve} \text{ 86} \rangle \equiv$

```
765 point piecewise_bezier_curve::evaluate(const double u) const {
766     if (_curves.size() == 0) return cagd::point(2);
767     double max_u = static_cast<double>(_curves.size());
768     if ((u < 0.) || (max_u < u)) return cagd::point(dimension());
769     size_t index;
770     if (u == max_u) {
771         index = static_cast<long>(u) - 1;
772     } else {
773         index = static_cast<long>(std::floor(u));
774     }
775     return _curves[index].evaluate(u);
776 }
777 point piecewise_bezier_curve::derivative(const double u) const {
778     if (_curves.size() == 0) return cagd::point(2);
779     double max_u = static_cast<double>(_curves.size());
780     if ((u < 0.) || (max_u < u)) return cagd::point(dimension());
781     size_t index;
782     if (u == max_u) {
783         index = static_cast<long>(u) - 1;
784     } else {
785         index = static_cast<long>(std::floor(u));
786     }
787     return _curves[index].derivative(u);
788 }
```

이 코드는 75번 마디에서 사용된다.

87.  $\langle \text{Methods of } \texttt{piecewise\_bezier\_curve} \text{ 77} \rangle + \equiv$

```
789 public:
790     point evaluate(const double) const;
791     point derivative(const double) const;
```

88. `piecewise_bezier_curve` 타입의 PostScript 출력을 위한 method들이다.

〈PostScript output of `piecewise_bezier_curve` 88〉≡

```

792 void piecewise_bezier_curve::write_curve_in_postscript(
793     psf &ps_file,
794     unsigned step,
795     float line_width,
796     int x, int y,
797     float magnification
798 ) const {
799     ios_base::fmtflags previous_options = ps_file.flags();
800     ps_file.precision(4);
801     ps_file.setf(ios_base::fixed, ios_base::floatfield);
802     for (const_curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
803         ps_file << "newpath" << endl << "[ ]0 setdash" << line_width << " setlinewidth" << endl;
804         point pt = magnification * (crv->evaluate(0));
805         ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
806         for (size_t i = 1; i ≤ step; i++) {
807             double t = double(i)/double(step);
808             pt = magnification * (crv->evaluate(t));
809             ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
810         }
811         ps_file << "stroke" << endl;
812     }
813     ps_file.flags(previous_options);
814 }

815 void piecewise_bezier_curve::write_control_polygon_in_postscript(
816     psf &ps_file,
817     float line_width,
818     int x, int y,
819     float magnification
820 ) const {
821     ios_base::fmtflags previous_options = ps_file.flags();
822     ps_file.precision(4);
823     ps_file.setf(ios_base::fixed, ios_base::floatfield);
824     for (const_curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
825         ps_file << "newpath" << endl;
826         ps_file << "[ ]0 setdash" << .5 * line_width << " setlinewidth" << endl;
827         point pt = magnification * (crv->ctrl_pts(0));
828         ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
829         for (size_t i = 1; i ≠ crv->ctrl_pts_size(); ++i) {

```

```

830     pt = magnification * (crv->ctrl_pts(i));
831     ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
832 }
833 ps_file << "stroke" << endl;
834 }
835 ps_file.flags(previous_options);
836 }

837 void piecewise_bezier_curve::write_control_points_in_postscript(
838     psf &ps_file,
839     float line_width,
840     int x, int y,
841     float magnification
842 ) const {
843     ios_base::fmtflags previous_options = ps_file.flags();
844     ps_file.precision(4);
845     ps_file.setf(ios_base::fixed, ios_base::floatfield);
846     for (const_curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
847         ps_file << "0\setgray" << endl;
848         ps_file << "newpath" << endl;
849         point pt = magnification * (crv->ctrl_pts(0));
850         ps_file << pt(x) << "\t" << pt(y) << "\t";
851         ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
852         ps_file << "closepath" << endl;
853         ps_file << "fill\stroke" << endl;
854         if (crv->ctrl_pts_size() > 2) {
855             for (size_t i = 1; i != crv->ctrl_pts_size() - 1; ++i) {
856                 ps_file << "newpath" << endl;
857                 pt = magnification * (crv->ctrl_pts(i));
858                 ps_file << pt(x) << "\t" << pt(y) << "\t";
859                 ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
860                 ps_file << "closepath" << endl;
861                 ps_file << line_width << "\t" << "setlinewidth" << endl;
862                 ps_file << "stroke" << endl;
863             }
864             ps_file << "0\setgray" << endl;
865             ps_file << "newpath" << endl;
866             pt = magnification * (crv->ctrl_pts(crv->ctrl_pts_size() - 1));
867             ps_file << pt(x) << "\t" << pt(y) << "\t";
868             ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
869             ps_file << "closepath" << endl;
870             ps_file << "fill\stroke" << endl;

```

```

871         }
872     }
873     ps_file.flags(previous_options);
874 }

```

이 코드는 75번 마디에서 사용된다.

### 89. 〈Methods of `piecewise_bezier_curve` 77〉 +≡

```

875 public:
876     void write_curve_in_postscript(
877         psf &, unsigned, float, int x = 1, int y = 2,
878         float magnification = 1.) const;
879     void write_control_polygon_in_postscript(
880         psf &, float, int x = 1, int y = 2,
881         float magnification = 1.) const;
882     void write_control_points_in_postscript(
883         psf &, float, int x = 1, int y = 2,
884         float magnification = 1.) const;

```



**90.** Test of `piecewise_bezier_curve` type. `piecewise_bezier_curve` 객체를 통한 `bezier` 곡선의 생성과 조작을 보여준다. Traditional Chinese character 중 하나를 골라 글자의 외곽선을 여러개의 Bézier 곡선으로 근사화한다. 곡선의 차수는 3차부터 7차까지 다양하게 섞여 있다. Bézier 곡선들을 하나의 `piecewise_bezier_curve` 객체로 묶은 후, 원래 형상을 PostScript 파일로 기술한다. 그 다음에 `piecewise_bezier_curve` 객체의 차수, 즉 그것을 구성하는 Bézier 곡선들 중 가장 높은 차수에 맞춰 degree elevation을 수행하고 결과를 다른 PostScript 파일에 기술한다. 마지막으로 모든 곡선 조각들을 다시 3차 Bézier 곡선으로 차수를 낮춘 후, 또 다른 PostScript 파일에 기술한다.

〈Test routines 26〉 +=

```

885     print_title("piecewise_bezier_curve");
886     {
887         piecewise_bezier_curve curves;
888         vector<point> ctrl_pts;
889         〈Build-up 3rd brush 91〉;
890         〈Build-up 2nd, 4th, and 5th brush 92〉;
891         〈Build-up 1st brush 93〉;
892         〈Build-up 6th, 7th, 8th, and 9th brush (outer part) 94〉;
893         〈Build-up 6th, 7th, 8th, and 9th brush (inner part) 95〉;
894         psf file = create_postscript_file("untouched.ps");    /* Draw original outline. */
895         curves.write_curve_in_postscript(file, 100, 1.);
896         curves.write_control_polygon_in_postscript(file, 1.);
897         curves.write_control_points_in_postscript(file, 1.);
898         close_postscript_file(file, true);
899         unsigned long deg = curves.degree();    /* Degree elevation. */
900         curves.elevate_degree(deg);
901         file = create_postscript_file("degree_elevated.ps");
902         curves.write_curve_in_postscript(file, 100, 1.);
903         curves.write_control_polygon_in_postscript(file, 1.);
904         curves.write_control_points_in_postscript(file, 1.);
905         close_postscript_file(file, true);
906         curves.reduce_degree(3);    /* Degree reduction. */
907         file = create_postscript_file("degree_reduced.ps");
908         curves.write_curve_in_postscript(file, 100, 1.);
909         curves.write_control_polygon_in_postscript(file, 1.);
910         curves.write_control_points_in_postscript(file, 1.);
911         close_postscript_file(file, true);
912     }

```

**91.**    〈 Build-up 3rd brush 91 〉  $\equiv$ 

```

913      ctrl_pts.push_back(point({183,416}));      /* 1st curve */
914      ctrl_pts.push_back(point({184,415}));
915      ctrl_pts.push_back(point({185,413}));
916      ctrl_pts.push_back(point({186,412}));
917      ctrl_pts.push_back(point({186,411}));
918      ctrl_pts.push_back(point({186,409}));
919      ctrl_pts.push_back(point({184,405}));
920      ctrl_pts.push_back(point({180,401}));
921      curves.push_back(bezier(ctrl_pts));
922      ctrl_pts.clear();
923      ctrl_pts.push_back(point({180,401}));      /* 2nd curve */
924      ctrl_pts.push_back(point({176,397}));
925      ctrl_pts.push_back(point({172,394}));
926      ctrl_pts.push_back(point({154,359}));
927      ctrl_pts.push_back(point({140,333}));
928      ctrl_pts.push_back(point({126,312}));
929      curves.push_back(bezier(ctrl_pts));
930      ctrl_pts.clear();
931      ctrl_pts.push_back(point({126,312}));      /* 3rd curve */
932      ctrl_pts.push_back(point({103,278}));
933      ctrl_pts.push_back(point({79,252}));
934      ctrl_pts.push_back(point({53,235}));
935      curves.push_back(bezier(ctrl_pts));
936      ctrl_pts.clear();
937      ctrl_pts.push_back(point({53,235}));      /* 4th curve */
938      ctrl_pts.push_back(point({46,230}));
939      ctrl_pts.push_back(point({42,228}));
940      ctrl_pts.push_back(point({37,231}));
941      curves.push_back(bezier(ctrl_pts));
942      ctrl_pts.clear();
943      ctrl_pts.push_back(point({37,231}));      /* 5th curve */
944      ctrl_pts.push_back(point({37,223}));
945      ctrl_pts.push_back(point({39,236}));
946      ctrl_pts.push_back(point({43,243}));
947      ctrl_pts.push_back(point({45,246}));
948      ctrl_pts.push_back(point({62,266}));
949      ctrl_pts.push_back(point({76,288}));
950      ctrl_pts.push_back(point({89,313}));
951      curves.push_back(bezier(ctrl_pts));
952      ctrl_pts.clear();
953      ctrl_pts.push_back(point({89,313}));      /* 6th curve */

```

```

954     ctrl_pts.push_back(point({102, 339}));
955     ctrl_pts.push_back(point({115, 369}));
956     ctrl_pts.push_back(point({127, 404}));
957     curves.push_back(bezier(ctrl_pts));
958     ctrl_pts.clear();
959     ctrl_pts.push_back(point({127, 404}));    /* 7th curve */
960     ctrl_pts.push_back(point({117, 400}));
961     ctrl_pts.push_back(point({107, 395}));
962     ctrl_pts.push_back(point({97, 392}));
963     curves.push_back(bezier(ctrl_pts));
964     ctrl_pts.clear();
965     ctrl_pts.push_back(point({97, 392}));    /* 8th curve */
966     ctrl_pts.push_back(point({86, 388}));
967     ctrl_pts.push_back(point({81, 386}));
968     ctrl_pts.push_back(point({74, 386}));
969     ctrl_pts.push_back(point({67, 388}));
970     ctrl_pts.push_back(point({57, 394}));
971     curves.push_back(bezier(ctrl_pts));
972     ctrl_pts.clear();
973     ctrl_pts.push_back(point({57, 394}));    /* 9th curve */
974     ctrl_pts.push_back(point({46, 399}));
975     ctrl_pts.push_back(point({41, 403}));
976     ctrl_pts.push_back(point({42, 406}));
977     ctrl_pts.push_back(point({43, 407}));
978     ctrl_pts.push_back(point({44, 407}));
979     curves.push_back(bezier(ctrl_pts));
980     ctrl_pts.clear();
981     ctrl_pts.push_back(point({44, 407}));    /* 10th curve */
982     ctrl_pts.push_back(point({46, 408}));
983     ctrl_pts.push_back(point({50, 409}));
984     ctrl_pts.push_back(point({68, 409}));
985     ctrl_pts.push_back(point({81, 410}));
986     ctrl_pts.push_back(point({94, 413}));
987     curves.push_back(bezier(ctrl_pts));
988     ctrl_pts.clear();
989     ctrl_pts.push_back(point({94, 413}));    /* 11th curve */
990     ctrl_pts.push_back(point({106, 416}));
991     ctrl_pts.push_back(point({115, 419}));
992     ctrl_pts.push_back(point({123, 425}));
993     ctrl_pts.push_back(point({127, 428}));
994     ctrl_pts.push_back(point({135, 439}));
995     ctrl_pts.push_back(point({139, 441}));

```

```
996     ctrl_pts.push_back(point({143, 441}));
997     curves.push_back(bezier(ctrl_pts));
998     ctrl_pts.clear();
999     ctrl_pts.push_back(point({143, 441}));     /* 12th curve */
1000     ctrl_pts.push_back(point({148, 441}));
1001     ctrl_pts.push_back(point({156, 438}));
1002     ctrl_pts.push_back(point({169, 429}));
1003     ctrl_pts.push_back(point({175, 423}));
1004     ctrl_pts.push_back(point({183, 416}));
1005     curves.push_back(bezier(ctrl_pts));
1006     ctrl_pts.clear();
```

이 코드는 90번 마디에서 사용된다.

**92.**  $\langle$  Build-up 2nd, 4th, and 5th brush 92  $\rangle \equiv$

```

1007  ctrl_pts.push_back(point({545, 226}));    /* 13th curve */
1008  ctrl_pts.push_back(point({547, 225}));
1009  ctrl_pts.push_back(point({550, 223}));
1010  ctrl_pts.push_back(point({554, 217}));
1011  ctrl_pts.push_back(point({555, 215}));
1012  ctrl_pts.push_back(point({555, 211}));
1013  ctrl_pts.push_back(point({547, 208}));
1014  ctrl_pts.push_back(point({532, 206}));
1015  curves.push_back(bezier(ctrl_pts));
1016  ctrl_pts.clear();
1017  ctrl_pts.push_back(point({532, 206}));    /* 14th curve */
1018  ctrl_pts.push_back(point({517, 204}));
1019  ctrl_pts.push_back(point({501, 203}));
1020  ctrl_pts.push_back(point({482, 203}));
1021  curves.push_back(bezier(ctrl_pts));
1022  ctrl_pts.clear();
1023  ctrl_pts.push_back(point({482, 203}));    /* 15th curve */
1024  ctrl_pts.push_back(point({460, 203}));
1025  ctrl_pts.push_back(point({430, 217}));
1026  ctrl_pts.push_back(point({392, 247}));
1027  curves.push_back(bezier(ctrl_pts));
1028  ctrl_pts.clear();
1029  ctrl_pts.push_back(point({392, 247}));    /* 16th curve */
1030  ctrl_pts.push_back(point({329, 299}));
1031  ctrl_pts.push_back(point({265, 366}));
1032  ctrl_pts.push_back(point({230, 410}));
1033  curves.push_back(bezier(ctrl_pts));
1034  ctrl_pts.clear();
1035  ctrl_pts.push_back(point({230, 410}));    /* 18th curve */
1036  ctrl_pts.push_back(point({230, 349}));
1037  ctrl_pts.push_back(point({230, 288}));
1038  ctrl_pts.push_back(point({230, 227}));
1039  curves.push_back(bezier(ctrl_pts));
1040  ctrl_pts.clear();
1041  ctrl_pts.push_back(point({230, 227}));    /* 19th curve */
1042  ctrl_pts.push_back(point({230, 215}));
1043  ctrl_pts.push_back(point({228, 204}));
1044  ctrl_pts.push_back(point({224, 193}));
1045  curves.push_back(bezier(ctrl_pts));
1046  ctrl_pts.clear();
1047  ctrl_pts.push_back(point({224, 193}));    /* 20th curve */

```

```

1048   ctrl_pts.push_back(point({219, 178}));
1049   ctrl_pts.push_back(point({211, 171}));
1050   ctrl_pts.push_back(point({196, 171}));
1051   ctrl_pts.push_back(point({190, 176}));
1052   ctrl_pts.push_back(point({174, 201}));
1053   ctrl_pts.push_back(point({169, 208}));
1054   ctrl_pts.push_back(point({169, 209}));
1055   curves.push_back(bezier(ctrl_pts));
1056   ctrl_pts.clear();
1057   ctrl_pts.push_back(point({169, 209})); /* 21st curve */
1058   ctrl_pts.push_back(point({160, 217}));
1059   ctrl_pts.push_back(point({152, 226}));
1060   ctrl_pts.push_back(point({135, 243}));
1061   ctrl_pts.push_back(point({131, 248}));
1062   ctrl_pts.push_back(point({131, 250}));
1063   curves.push_back(bezier(ctrl_pts));
1064   ctrl_pts.clear();
1065   ctrl_pts.push_back(point({131, 250})); /* 22nd curve */
1066   ctrl_pts.push_back(point({133, 252}));
1067   ctrl_pts.push_back(point({135, 253}));
1068   ctrl_pts.push_back(point({140, 253}));
1069   ctrl_pts.push_back(point({149, 251}));
1070   ctrl_pts.push_back(point({163, 246}));
1071   curves.push_back(bezier(ctrl_pts));
1072   ctrl_pts.clear();
1073   ctrl_pts.push_back(point({163, 246})); /* 23rd curve */
1074   ctrl_pts.push_back(point({170, 243}));
1075   ctrl_pts.push_back(point({175, 242}));
1076   ctrl_pts.push_back(point({188, 242}));
1077   ctrl_pts.push_back(point({192, 247}));
1078   ctrl_pts.push_back(point({192, 258}));
1079   curves.push_back(bezier(ctrl_pts));
1080   ctrl_pts.clear();
1081   ctrl_pts.push_back(point({192, 258})); /* 24th curve */
1082   ctrl_pts.push_back(point({192, 342}));
1083   ctrl_pts.push_back(point({192, 426}));
1084   ctrl_pts.push_back(point({192, 509}));
1085   curves.push_back(bezier(ctrl_pts));
1086   ctrl_pts.clear();
1087   ctrl_pts.push_back(point({192, 509})); /* 25th curve */
1088   ctrl_pts.push_back(point({192, 515}));
1089   ctrl_pts.push_back(point({192, 519}));

```

```

1090     ctrl_pts.push_back(point({189, 525}));
1091     ctrl_pts.push_back(point({186, 526}));
1092     ctrl_pts.push_back(point({175, 526}));
1093     ctrl_pts.push_back(point({166, 523}));
1094     ctrl_pts.push_back(point({154, 517}));
1095     curves.push_back(bezier(ctrl_pts));
1096     ctrl_pts.clear();
1097     ctrl_pts.push_back(point({154, 517}));    /* 26th curve */
1098     ctrl_pts.push_back(point({143, 511}));
1099     ctrl_pts.push_back(point({134, 508}));
1100     ctrl_pts.push_back(point({124, 508}));
1101     ctrl_pts.push_back(point({117, 510}));
1102     ctrl_pts.push_back(point({107, 512}));
1103     curves.push_back(bezier(ctrl_pts));
1104     ctrl_pts.clear();
1105     ctrl_pts.push_back(point({107, 512}));    /* 27th curve */
1106     ctrl_pts.push_back(point({98, 515}));
1107     ctrl_pts.push_back(point({93, 518}));
1108     ctrl_pts.push_back(point({93, 520}));
1109     curves.push_back(bezier(ctrl_pts));
1110     ctrl_pts.clear();
1111     ctrl_pts.push_back(point({93, 520}));    /* 28th curve */
1112     ctrl_pts.push_back(point({93, 522}));
1113     ctrl_pts.push_back(point({95, 523}));
1114     ctrl_pts.push_back(point({103, 526}));
1115     ctrl_pts.push_back(point({107, 527}));
1116     ctrl_pts.push_back(point({110, 527}));
1117     curves.push_back(bezier(ctrl_pts));
1118     ctrl_pts.clear();
1119     ctrl_pts.push_back(point({110, 527}));    /* 29th curve */
1120     ctrl_pts.push_back(point({122, 530}));
1121     ctrl_pts.push_back(point({134, 534}));
1122     ctrl_pts.push_back(point({154, 541}));
1123     ctrl_pts.push_back(point({165, 545}));
1124     ctrl_pts.push_back(point({180, 552}));
1125     ctrl_pts.push_back(point({183, 555}));
1126     ctrl_pts.push_back(point({188, 560}));
1127     curves.push_back(bezier(ctrl_pts));
1128     ctrl_pts.clear();
1129     ctrl_pts.push_back(point({188, 560}));    /* 30th curve */
1130     ctrl_pts.push_back(point({192, 566}));
1131     ctrl_pts.push_back(point({196, 568}));

```

```

1132   ctrl_pts.push_back(point({204, 568}));
1133   ctrl_pts.push_back(point({213, 562}));
1134   ctrl_pts.push_back(point({241, 537}));
1135   ctrl_pts.push_back(point({248, 529}));
1136   ctrl_pts.push_back(point({248, 524}));
1137   curves.push_back(bezier(ctrl_pts));
1138   ctrl_pts.clear();
1139   ctrl_pts.push_back(point({248, 524}));   /* 31st curve */
1140   ctrl_pts.push_back(point({248, 521}));
1141   ctrl_pts.push_back(point({246, 517}));
1142   ctrl_pts.push_back(point({238, 506}));
1143   ctrl_pts.push_back(point({235, 502}));
1144   ctrl_pts.push_back(point({235, 501}));
1145   curves.push_back(bezier(ctrl_pts));
1146   ctrl_pts.clear();
1147   ctrl_pts.push_back(point({235, 501}));   /* 32nd curve */
1148   ctrl_pts.push_back(point({231, 481}));
1149   ctrl_pts.push_back(point({230, 457}));
1150   ctrl_pts.push_back(point({230, 437}));
1151   curves.push_back(bezier(ctrl_pts));
1152   ctrl_pts.clear();
1153   ctrl_pts.push_back(point({230, 437}));   /* 33rd curve */
1154   ctrl_pts.push_back(point({232.5, 433}));
1155   ctrl_pts.push_back(point({235, 429}));
1156   curves.push_back(bezier(ctrl_pts));
1157   ctrl_pts.clear();
1158   ctrl_pts.push_back(point({235, 429}));   /* 34th curve */
1159   ctrl_pts.push_back(point({256, 452}));
1160   ctrl_pts.push_back(point({280, 486}));
1161   ctrl_pts.push_back(point({295, 515}));
1162   curves.push_back(bezier(ctrl_pts));
1163   ctrl_pts.clear();
1164   ctrl_pts.push_back(point({295, 515}));   /* 35th curve */
1165   ctrl_pts.push_back(point({295, 519}));
1166   ctrl_pts.push_back(point({296, 523}));
1167   ctrl_pts.push_back(point({298, 530}));
1168   ctrl_pts.push_back(point({301, 531}));
1169   ctrl_pts.push_back(point({312, 531}));
1170   ctrl_pts.push_back(point({321, 528}));
1171   ctrl_pts.push_back(point({334, 520}));
1172   curves.push_back(bezier(ctrl_pts));
1173   ctrl_pts.clear();

```



```

1174   ctrl_pts.push_back(point({334, 520}));   /* 36th curve */
1175   ctrl_pts.push_back(point({347, 512}));
1176   ctrl_pts.push_back(point({354, 505}));
1177   ctrl_pts.push_back(point({354, 499}));
1178   curves.push_back(bezier(ctrl_pts));
1179   ctrl_pts.clear();
1180   ctrl_pts.push_back(point({354, 499}));   /* 37th curve */
1181   ctrl_pts.push_back(point({354, 496}));
1182   ctrl_pts.push_back(point({351, 493}));
1183   ctrl_pts.push_back(point({340, 487}));
1184   ctrl_pts.push_back(point({335, 484}));
1185   ctrl_pts.push_back(point({330, 482}));
1186   curves.push_back(bezier(ctrl_pts));
1187   ctrl_pts.clear();
1188   ctrl_pts.push_back(point({330, 482}));   /* 38th curve */
1189   ctrl_pts.push_back(point({304, 461}));
1190   ctrl_pts.push_back(point({274, 437}));
1191   ctrl_pts.push_back(point({243, 416}));
1192   curves.push_back(bezier(ctrl_pts));
1193   ctrl_pts.clear();
1194   ctrl_pts.push_back(point({243, 416}));   /* 39th curve */
1195   ctrl_pts.push_back(point({283, 370}));
1196   ctrl_pts.push_back(point({342, 325}));
1197   ctrl_pts.push_back(point({413, 283}));
1198   curves.push_back(bezier(ctrl_pts));
1199   ctrl_pts.clear();
1200   ctrl_pts.push_back(point({413, 283}));   /* 40th curve */
1201   ctrl_pts.push_back(point({456, 262}));
1202   ctrl_pts.push_back(point({523, 235}));
1203   ctrl_pts.push_back(point({545, 226}));
1204   curves.push_back(bezier(ctrl_pts));
1205   ctrl_pts.clear();

```

이 코드는 90번 마디에서 사용된다.

**93.** 〈Build-up 1st brush 93〉 ≡

```

1206  ctrl_pts.push_back(point({245, 638})); /* 41st curve */
1207  ctrl_pts.push_back(point({249, 633}));
1208  ctrl_pts.push_back(point({251, 625}));
1209  ctrl_pts.push_back(point({251, 614}));
1210  curves.push_back(bezier(ctrl_pts));
1211  ctrl_pts.clear();
1212  ctrl_pts.push_back(point({251, 614})); /* 42nd curve */
1213  ctrl_pts.push_back(point({251, 603}));
1214  ctrl_pts.push_back(point({247, 597}));
1215  ctrl_pts.push_back(point({240, 597}));
1216  curves.push_back(bezier(ctrl_pts));
1217  ctrl_pts.clear();
1218  ctrl_pts.push_back(point({240, 597})); /* 43rd curve */
1219  ctrl_pts.push_back(point({219, 608}));
1220  ctrl_pts.push_back(point({164, 651}));
1221  ctrl_pts.push_back(point({151, 666}));
1222  curves.push_back(bezier(ctrl_pts));
1223  ctrl_pts.clear();
1224  ctrl_pts.push_back(point({151, 666})); /* 44th curve */
1225  ctrl_pts.push_back(point({152, 667}));
1226  ctrl_pts.push_back(point({153, 667}));
1227  ctrl_pts.push_back(point({155, 668}));
1228  ctrl_pts.push_back(point({156, 668}));
1229  ctrl_pts.push_back(point({157, 668}));
1230  curves.push_back(bezier(ctrl_pts));
1231  ctrl_pts.clear();
1232  ctrl_pts.push_back(point({157, 668})); /* 45th curve */
1233  ctrl_pts.push_back(point({189, 668}));
1234  ctrl_pts.push_back(point({224, 655}));
1235  ctrl_pts.push_back(point({245, 638}));
1236  curves.push_back(bezier(ctrl_pts));
1237  ctrl_pts.clear();

```

이 코드는 90번 마디에서 사용된다.

94. 〈Build-up 6th, 7th, 8th, and 9th brush (outer part) 94〉 ≡

```

1238  ctrl_pts.push_back(point({535, 598}));    /* 46th curve */
1239  ctrl_pts.push_back(point({537, 596}));
1240  ctrl_pts.push_back(point({539, 593}));
1241  ctrl_pts.push_back(point({539, 585}));
1242  ctrl_pts.push_back(point({537, 581}));
1243  ctrl_pts.push_back(point({529, 568}));
1244  ctrl_pts.push_back(point({526, 564}));
1245  ctrl_pts.push_back(point({526, 564}));
1246  curves.push_back(bezier(ctrl_pts));
1247  ctrl_pts.clear();
1248  ctrl_pts.push_back(point({526, 564}));    /* 47th curve */
1249  ctrl_pts.push_back(point({526, 507}));
1250  ctrl_pts.push_back(point({526, 451}));
1251  ctrl_pts.push_back(point({526, 394}));
1252  curves.push_back(bezier(ctrl_pts));
1253  ctrl_pts.clear();
1254  ctrl_pts.push_back(point({526, 394}));    /* 48th curve */
1255  ctrl_pts.push_back(point({527, 379}));
1256  ctrl_pts.push_back(point({528, 364}));
1257  ctrl_pts.push_back(point({529, 348}));
1258  curves.push_back(bezier(ctrl_pts));
1259  ctrl_pts.clear();
1260  ctrl_pts.push_back(point({529, 348}));    /* 49th curve */
1261  ctrl_pts.push_back(point({528, 331}));
1262  ctrl_pts.push_back(point({521, 312}));
1263  ctrl_pts.push_back(point({510, 307}));
1264  curves.push_back(bezier(ctrl_pts));
1265  ctrl_pts.clear();
1266  ctrl_pts.push_back(point({510, 307}));    /* 50th curve */
1267  ctrl_pts.push_back(point({502, 307}));
1268  ctrl_pts.push_back(point({496, 313}));
1269  ctrl_pts.push_back(point({489, 334}));
1270  ctrl_pts.push_back(point({488, 344}));
1271  ctrl_pts.push_back(point({487, 357}));
1272  curves.push_back(bezier(ctrl_pts));
1273  ctrl_pts.clear();
1274  ctrl_pts.push_back(point({487, 357}));    /* 51st curve */
1275  ctrl_pts.push_back(point({459, 356}));
1276  ctrl_pts.push_back(point({418, 352}));
1277  ctrl_pts.push_back(point({408, 347}));
1278  curves.push_back(bezier(ctrl_pts));

```

```

1279  ctrl_pts.clear();
1280  ctrl_pts.push_back(point({408, 347}));    /* 52nd curve */
1281  ctrl_pts.push_back(point({408, 335}));
1282  ctrl_pts.push_back(point({404, 330}));
1283  ctrl_pts.push_back(point({396, 330}));
1284  curves.push_back(bezier(ctrl_pts));
1285  ctrl_pts.clear();
1286  ctrl_pts.push_back(point({396, 330}));    /* 53rd curve */
1287  ctrl_pts.push_back(point({382, 336}));
1288  ctrl_pts.push_back(point({369, 360}));
1289  ctrl_pts.push_back(point({366, 377}));
1290  curves.push_back(bezier(ctrl_pts));
1291  ctrl_pts.clear();
1292  ctrl_pts.push_back(point({366, 377}));    /* 54th curve */
1293  ctrl_pts.push_back(point({367, 390}));
1294  ctrl_pts.push_back(point({371, 421}));
1295  ctrl_pts.push_back(point({372, 440}));
1296  curves.push_back(bezier(ctrl_pts));
1297  ctrl_pts.clear();
1298  ctrl_pts.push_back(point({372, 440}));    /* 55th curve */
1299  ctrl_pts.push_back(point({372, 435}));
1300  ctrl_pts.push_back(point({372, 439}));
1301  ctrl_pts.push_back(point({372, 554}));
1302  curves.push_back(bezier(ctrl_pts));
1303  ctrl_pts.clear();
1304  ctrl_pts.push_back(point({372, 554}));    /* 56th curve */
1305  ctrl_pts.push_back(point({372, 564}));
1306  ctrl_pts.push_back(point({360, 594}));
1307  ctrl_pts.push_back(point({355, 603}));
1308  ctrl_pts.push_back(point({353, 617}));
1309  ctrl_pts.push_back(point({358, 617}));
1310  curves.push_back(bezier(ctrl_pts));
1311  ctrl_pts.clear();
1312  ctrl_pts.push_back(point({358, 617}));    /* 57th curve */
1313  ctrl_pts.push_back(point({365, 617}));
1314  ctrl_pts.push_back(point({372, 615}));
1315  ctrl_pts.push_back(point({385, 607}));
1316  ctrl_pts.push_back(point({392, 603}));
1317  ctrl_pts.push_back(point({398, 600}));
1318  curves.push_back(bezier(ctrl_pts));
1319  ctrl_pts.clear();
1320  ctrl_pts.push_back(point({398, 600}));    /* 58th curve */

```

```

1321  ctrl_pts.push_back(point({417, 603}));
1322  ctrl_pts.push_back(point({443, 609}));
1323  ctrl_pts.push_back(point({463, 613}));
1324  curves.push_back(bezier(ctrl_pts));
1325  ctrl_pts.clear();
1326  ctrl_pts.push_back(point({463, 613})); /* 59th curve */
1327  ctrl_pts.push_back(point({470, 618}));
1328  ctrl_pts.push_back(point({480, 629}));
1329  ctrl_pts.push_back(point({487, 632}));
1330  curves.push_back(bezier(ctrl_pts));
1331  ctrl_pts.clear();
1332  ctrl_pts.push_back(point({487, 632})); /* 60th curve */
1333  ctrl_pts.push_back(point({499, 627}));
1334  ctrl_pts.push_back(point({520, 611}));
1335  ctrl_pts.push_back(point({535, 598}));
1336  curves.push_back(bezier(ctrl_pts));
1337  ctrl_pts.clear();

```

이 코드는 90번 마디에서 사용된다.

**95.**    〈 Build-up 6th, 7th, 8th, and 9th brush (inner part) 95〉  $\equiv$

```

1338   ctrl_pts.push_back(point({487, 378}));   /* 61st curve */
1339   ctrl_pts.push_back(point({487, 444}));
1340   ctrl_pts.push_back(point({487, 510}));
1341   ctrl_pts.push_back(point({487, 576}));
1342   curves.push_back(bezier(ctrl_pts));
1343   ctrl_pts.clear();
1344   ctrl_pts.push_back(point({487, 576}));   /* 62nd curve */
1345   ctrl_pts.push_back(point({487, 583}));
1346   ctrl_pts.push_back(point({486, 587}));
1347   ctrl_pts.push_back(point({484, 594}));
1348   ctrl_pts.push_back(point({480, 597}));
1349   ctrl_pts.push_back(point({473, 597}));
1350   curves.push_back(bezier(ctrl_pts));
1351   ctrl_pts.clear();
1352   ctrl_pts.push_back(point({473, 597}));   /* 63rd curve */
1353   ctrl_pts.push_back(point({454, 596}));
1354   ctrl_pts.push_back(point({428, 590}));
1355   ctrl_pts.push_back(point({408, 584}));
1356   curves.push_back(bezier(ctrl_pts));
1357   ctrl_pts.clear();
1358   ctrl_pts.push_back(point({408, 584}));   /* 64th curve */
1359   ctrl_pts.push_back(point({408, 553}));
1360   ctrl_pts.push_back(point({408, 523}));
1361   ctrl_pts.push_back(point({408, 492}));
1362   curves.push_back(bezier(ctrl_pts));
1363   ctrl_pts.clear();
1364   ctrl_pts.push_back(point({408, 492}));   /* 65th curve */
1365   ctrl_pts.push_back(point({420, 494}));
1366   ctrl_pts.push_back(point({447, 504}));
1367   ctrl_pts.push_back(point({464, 507}));
1368   curves.push_back(bezier(ctrl_pts));
1369   ctrl_pts.clear();
1370   ctrl_pts.push_back(point({464, 507}));   /* 66th curve */
1371   ctrl_pts.push_back(point({475, 507}));
1372   ctrl_pts.push_back(point({481, 504}));
1373   ctrl_pts.push_back(point({481, 489}));
1374   ctrl_pts.push_back(point({471, 482}));
1375   ctrl_pts.push_back(point({450, 478}));
1376   curves.push_back(bezier(ctrl_pts));
1377   ctrl_pts.clear();
1378   ctrl_pts.push_back(point({450, 478}));   /* 67th curve */

```

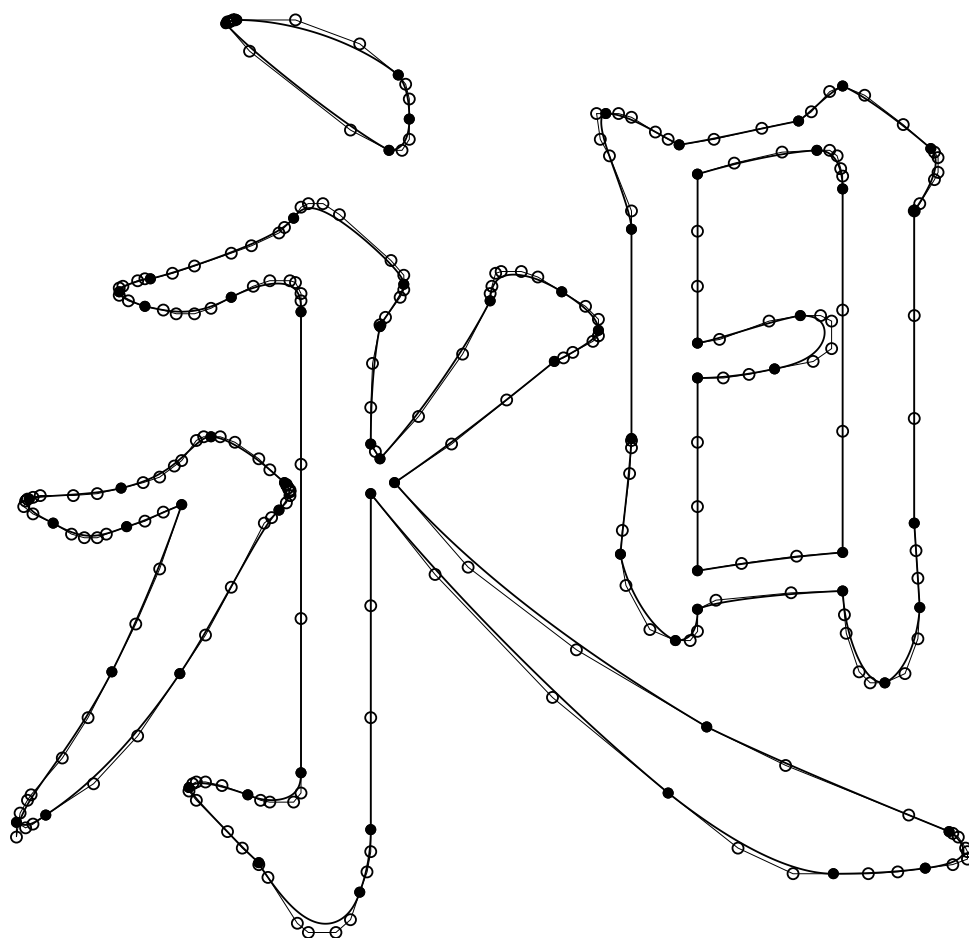
```

1379  ctrl_pts.push_back(point({436, 475}));
1380  ctrl_pts.push_back(point({422, 473}));
1381  ctrl_pts.push_back(point({408, 473}));
1382  curves.push_back(bezier(ctrl_pts));
1383  ctrl_pts.clear();
1384  ctrl_pts.push_back(point({408, 473})); /* 68th curve */
1385  ctrl_pts.push_back(point({408, 438}));
1386  ctrl_pts.push_back(point({408, 403}));
1387  ctrl_pts.push_back(point({408, 368}));
1388  curves.push_back(bezier(ctrl_pts));
1389  ctrl_pts.clear();
1390  ctrl_pts.push_back(point({408, 368})); /* 69th curve */
1391  ctrl_pts.push_back(point({432, 372}));
1392  ctrl_pts.push_back(point({462, 376}));
1393  ctrl_pts.push_back(point({487, 378}));
1394  curves.push_back(bezier(ctrl_pts));
1395  ctrl_pts.clear();

```

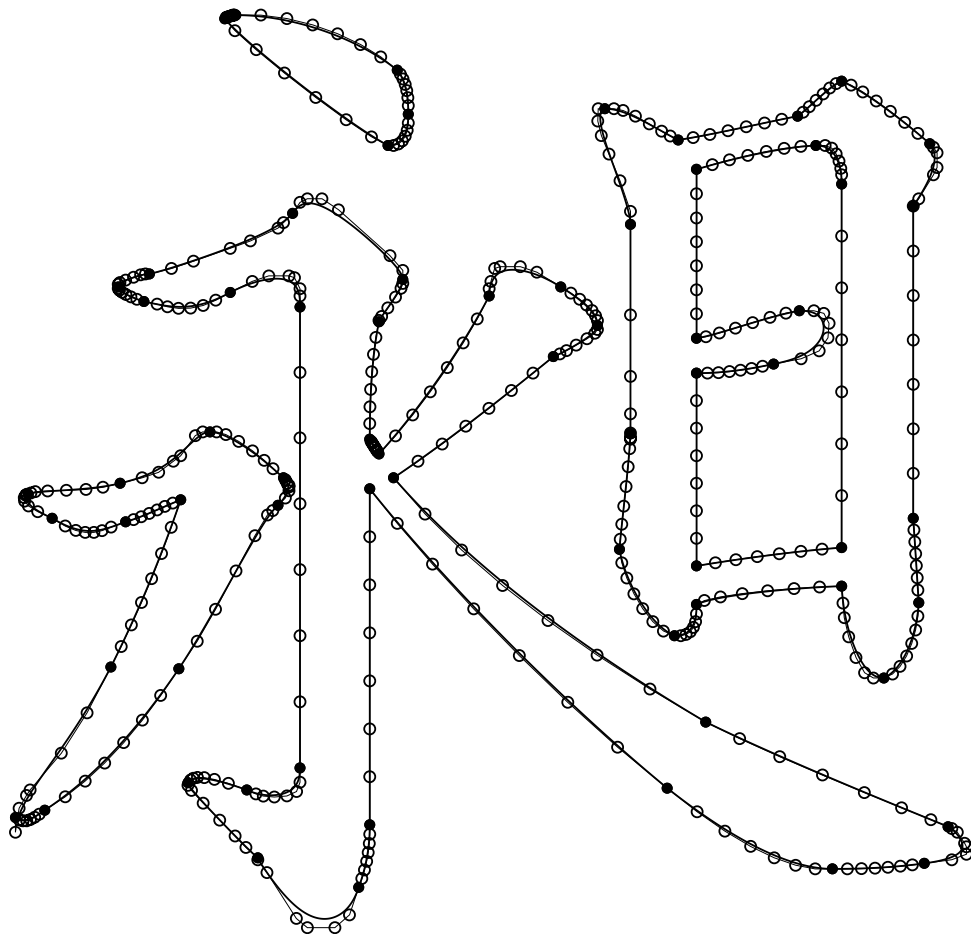
이 코드는 90번 마디에서 사용된다.

**96.** 예제 실행 결과. 첫 번째 그림은 traditional chinese 문자 중 하나를 골라 외곽선을 여러개의 Bézier 곡선으로 근사화한 것이다. 검은색 점은 각 곡선의 끝점을 나타내며, 흰 점은 중간의 컨트롤 포인트를, 가느다란 직선은 컨트롤 폴리곤을 나타낸다. 곡선의 차수는 3차부터 7차까지 다양하게 사용했다.

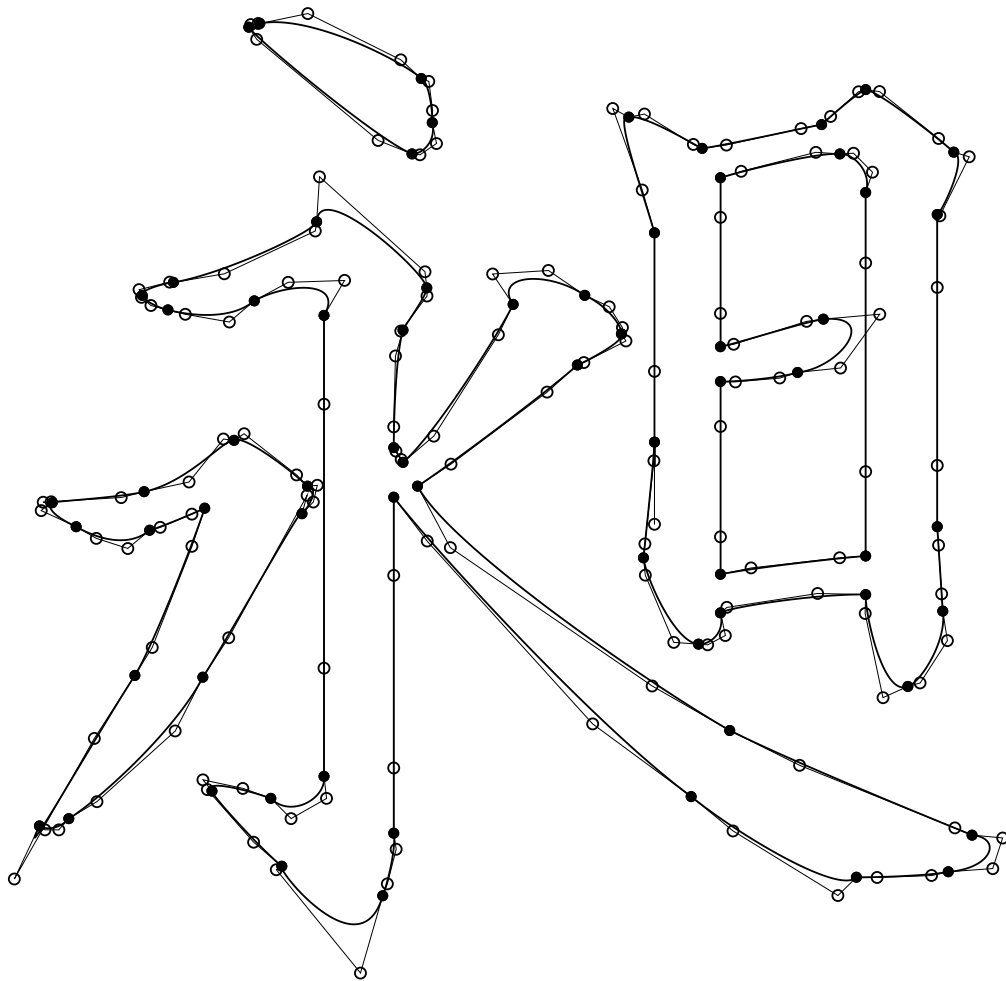




97. 아래 그림은 모든 곡선의 차수를 가장 차수가 높은 Bézier 곡선 조각의 차수에 맞춰 올린 것이다. 곡선의 차수를 올리더라도 곡선의 형상은 변화하지 않는다.



98. 아래 그림은 다시 모든 곡선의 차수를 3차로 낮춘 것이다. 곡선의 형상 변화를 최소화하는 컨트롤 포인트를 구했지만 완벽하게 동일한 모양을 얻은 것은 아니다. 특히 곡선의 컨트롤 폴리곤이 매우 들쭉 날쭉한 것에 유의해야 한다. 만약 곡선을 시간에 따라 애니메이션으로 그린다면 불규칙하게 배치된 컨트롤 포인트들이 문제를 일으킬 것이다. 따라서 Bézier 곡선의 차수를 낮추는 알고리즘을 로봇이나 기구의 동작 궤적에 적용할 때에는 각별한 주의가 필요하다.



**99. Cubic Spline Curve.**

〈Definition of **cubic\_spline** 99〉 ≡

```

1396     class cubic_spline : public curve {
1397         〈Data members of cubic_spline 100〉
1398         〈Enumerations of cubic_spline 143〉
1399         〈Methods of cubic_spline 103〉
1400     };

```

이 코드는 2번 마디에서 사용된다.

**100. cubic\_spline** 타입은 spline 곡선의 knot sequence를 data member로 갖는다. 컨트롤 포인트들을 저장하는 멤버는 **curve** 타입으로부터 상속받는다. 그리고 knot sequence를 반복문에서 간편하게 지칭하기 위한 iterator들의 타입을 선언한다. OpenCL을 이용해서 곡선상의 점들을 한꺼번에 계산하기 위한 **mpoi** 타입의 객체를 멤버로 갖는다.

〈Data members of **cubic\_spline** 100〉 ≡

```

1401     protected:
1402         vector<double> _knot_sqnc;
1403         mutable mpoi _mp;
1404         size_t _kernel_id;
1405     protected:
1406         typedef vector<double>::iterator knot_itr;
1407         typedef vector<double>::const_iterator const_knot_itr;

```

이 코드는 99번 마디에서 사용된다.

**101. cubic\_spline** 타입의 method들은 다음과 같다.

〈Implementation of **cubic\_spline** 101〉 ≡

```

1408     〈Constructors and destructor of cubic_spline 102〉
1409     〈Properties of cubic_spline 104〉
1410     〈Operators of cubic_spline 106〉
1411     〈Description of cubic_spline 108〉
1412     〈Evaluation and derivative of cubic_spline 110〉
1413     〈Methods for interpolation of cubic_spline 144〉
1414     〈Methods to obtain a bezier curve for a segment of cubic_spline 168〉
1415     〈Methods to calculate curvature of cubic_spline 174〉
1416     〈Methods for knot insertion and removal of cubic_spline 176〉
1417     〈Methods for PostScript output of cubic_spline 190〉
1418     〈Miscellaneous methods of cubic_spline 116〉

```

이 코드는 3번 마디에서 사용된다.

**102.** 다른 **cubic\_spline** 객체가 주어졌을 때 그것을 복제하는 복사생성자와, 그리고 (당연하게도) knot sequence와 control point들이 주어졌을 때 그것에 상응하는 곡선을 생성하는 constructor를 정의한다.

⟨Constructors and destructor of **cubic\_spline** 102⟩ ≡

```

1419   cubic_spline::cubic_spline(const cubic_spline &src)
1420       : curve(src),
1421         _knot_sqnc(src._knot_sqnc),
1422         _mp(src._mp),
1423         _kernel_id(src._kernel_id) {}

1424   cubic_spline::cubic_spline(const vector<double> &knots, const vector<point> &pts)
1425       : curve(pts),
1426         _mp("./cspline.cl"),
1427         _knot_sqnc(knots),
1428         _kernel_id(_mp.create_kernel("evaluate_crv")) {}

1429   cubic_spline::~cubic_spline() {}

```

146번 마디도 살펴보라.

이 코드는 101번 마디에서 사용된다.

**103.**

⟨Methods of **cubic\_spline** 103⟩ ≡

```

1430   public:
1431   cubic_spline() = delete ;
1432   cubic_spline(const cubic_spline &);
1433   cubic_spline(const vector<double> &, const vector<point> &);
1434   virtual ~cubic_spline();

```

105, 107, 109, 111, 113, 117, 119, 123, 125, 127, 129, 145, 147, 173, 175, 181, 183, 189, 191번 마디도 살펴보라.

이 코드는 99번 마디에서 사용된다.

**104.** **cubic\_spline** 객체의 대표적인 property는 차원과 차수다. 또한 knot sequence와 control point를 반환하는 method도 정의한다.

⟨Properties of **cubic\_spline** 104⟩ ≡

```

1435   unsigned long cubic_spline::degree() const {
1436       return 3;
1437   }

1438   vector<double> cubic_spline::knot_sequence() const {
1439       return _knot_sqnc;
1440   }

1441   vector<point> cubic_spline::control_points() const {
1442       return _ctrl_pts;
1443   }

```

이 코드는 101번 마디에서 사용된다.

**105.** 〈Methods of `cubic_spline` 103〉 +≡

```

1444 public:
1445     unsigned long degree() const;
1446     vector<double> knot_sequence() const;
1447     vector<point> control_points() const;

```

**106.** Operators of `cubic_spline`.〈Operators of `cubic_spline` 106〉 ≡

```

1448     cubic_spline &cubic_spline::operator=(const cubic_spline &crv) {
1449         curve::operator=(crv);
1450         _knot_sqnc = crv._knot_sqnc;
1451         _mp = crv._mp;
1452         _kernel_id = crv._kernel_id;
1453         return *this;
1454     }

```

이 코드는 101번 마디에서 사용된다.

**107.** 〈Methods of `cubic_spline` 103〉 +≡

```

1455 public:
1456     cubic_spline &operator=(const cubic_spline &);

```

**108.** Debugging을 위한 method를 정의한다.〈Description of `cubic_spline` 108〉 ≡

```

1457     string cubic_spline::description() const {
1458         stringstream buffer;
1459         buffer << curve::description();
1460         buffer << "Knot Sequence:" << endl;
1461         for (size_t i = 0; i < _knot_sqnc.size(); i++) {
1462             buffer << "    " << _knot_sqnc[i] << endl;
1463         }
1464         return buffer.str();
1465     }

```

이 코드는 101번 마디에서 사용된다.

**109.** 〈Methods of `cubic_spline` 103〉 +≡

```

1466 public:
1467     string description() const;

```

**110.** Cubic spline 곡선 위의 점은 잘 알려진바와 같이 de Boor 알고리즘으로 계산한다. Degree  $n$ 이고  $L$  개의 다항함수 조각(polynomial segments)으로 이루어진 B-spline 곡선은  $L + 2n - 1$ 개의 nondecreasing knot sequence

$$u_0, \dots, \underbrace{u_{n-1}, \dots, u_{L+n-1}}_{\text{domain knots}}, \dots, u_{L+2n-2}$$

를 갖는다. 이 때, 앞과 뒤 각각  $n$ 개씩의 knots에서는 곡선이 정의되지 않고, 가운데의  $L + 1$ 개의 knots에서 곡선이 정의되기에  $[u_{n-1}, \dots, u_{L+n-1}]$ 를 domain knots라 부른다.

이 때,  $n + L$ 개의 Greville abscissas

$$\xi_i = \frac{1}{n}(u_i + \dots + u_{i+n-1}); \quad i = 0, \dots, L + n - 1$$

에 control point들이 대응된다. 이는 functional spline을 생각하면 좀 더 쉽게 이해되는데, 점  $(\xi_i, d_i)$ ;  $i = 0, \dots, L + n - 1$ 들이 다각형  $P$ 를 이루고, de Boor algorithm은 이 다각형으로부터 반복적인 piecewise linear interpolation을 수행하여 곡선상의 점을 구하는 것이다.

구체적으로, degree  $n$ 인 B-spline 곡선의 knot sequence  $u_j$ 와 control points  $d_i$ 가 있을때,  $u \in [u_I, u_{I+1}) \subset [u_{n-1}, u_{L+n-1}]$ 를 만족하는  $u$ 에 대응하는 곡선상의 점은,  $k = 1, \dots, n - r$ ,  $i = I - n + k + 1, \dots, I - r + 1$ 에 대하여

$$d_i^k(u) = \frac{u_{i+n-k} - u}{u_{i+n-k} - u_{i-1}} d_{i-1}^{k-1}(u) + \frac{u - u_{i-1}}{u_{i+n-k} - u_{i-1}} d_i^{k-1}(u)$$

을 반복적으로 계산한 결과

$$d_{I-r+1}^{n-r}(u)$$

이다. 이때,  $r$ 은  $u$ 가 knot sequence 중 하나의 값일 때, 그것의 중첩도(multiplicity)이며, 특정한 knot sequence 값이 아니면 0으로 둔다. 위 점화식의 초기조건은

$$d_i^0(u) = d_i$$

로 둔다. Knot sequence에 해당하지 않는  $u \in [u_I, u_{I+1}]$ 에 대한 de Boor 알고리즘을 그림으로 표시하면 다음과 같다:

$$\begin{array}{ccccc} & d_{I-n+1} & & & \\ & d_{I-n+2} & d_{I-n+2}^1 & & \\ & \vdots & \vdots & \ddots & \\ & d_I & d_I^1 & \cdots & d_I^{n-1} \\ d_{I+1} & d_{I+1}^1 & \cdots & d_{I+1}^{n-1} & d_{I+1}^n \end{array}$$

`evaluate()` method는 세 가지 종류가 있다. 첫 번째는 evaluation abscissa  $u$ 와 그것이 속하는 구간에 대한 index  $I$ 를 입력으로 받는 method다. Index  $I$ 는 de Boor 알고리즘을 적용하기 위하여 반드시 필요한 것이지만, 대체로 곡선상의 점을 계산하기 위하여 일일이  $I$ 까지 알아내고 그것을 함께 인자로 전달하는 것은 번거로운 일이다. 따라서 두 번째 method는 evaluation abscissa  $u$ 만 인자로 전달 받으며, `find_index_in_knot_sequence()` 함수를 호출해서  $u[I] \leq u < u[I + 1]$ 을 만족하는 정수  $I$ 를 찾는다. 끝으로 세 번째 method는 OpenCL을 이용해서 주어진 간격 수로 evaluation abscissa 범위를 등간격으로 나눈 후, 그 값들에 대응하는 곡선상의 점들을 한꺼번에 계산한다.

첫 번째와 두 번째 method는 de Casteljau의 repeated linear interpolation algorithm의 일반화된 version 이라고 이해할 수 있으므로 자세한 설명은 생략한다.

⟨Evaluation and derivative of **cubic\_spline** 110⟩ ≡

```

1468   point cubic_spline::evaluate(const double u, unsigned long I) const {
1469       const unsigned long n = 3;      /* Degree of cubic spline. */
1470       vector<point> tmp;
1471       for (size_t i = I - n + 1; i <= I + 2; i++) {
1472           tmp.push_back(_ctrl_pts[i]);
1473       }
1474       long shifter = I - n + 1;
1475       for (size_t k = 1; k <= n + 1; k++) {
1476           for (size_t i = I + 1; i <= I - n + k; i++) {
1477               double t1 = (_knot_sqnc[i + n - k] - u) / (_knot_sqnc[i + n - k] - _knot_sqnc[i - 1]);
1478               double t2 = 1.0 - t1;
1479               tmp[i - shifter] = t1 * tmp[i - shifter - 1] + t2 * tmp[i - shifter];
1480           }
1481       }
1482       return tmp[I - shifter + 1];
1483   }
1484   point cubic_spline::evaluate(const double u) const {
1485       return evaluate(u, find_index_in_knot_sequence(u));
1486   }

```

112, 118번 마디도 살펴보자.

이 코드는 101번 마디에서 사용된다.

111. <Methods of cubic\_spline 103> +=

```

1487   public:
1488       point evaluate(const double, unsigned long) const;
1489       point evaluate(const double) const;

```

**112.** Knot sequence의 domain knots 범위를  $N - 1$ 개의 등간격으로 나누어 곡선위의  $N$ 개의 점을 한번에 계산하는 method를 구현한다. 즉, 곡선을  $N - 1$ 개의 작은 선분 조각들로 근사화하는 셈이다. 이 method는 계산할 점의 갯수를 입력인자  $N$ 으로 받으며, 계산 결과를 **vector<point>** 타입으로 반환한다.

Kernel에서 계산한  $m$ 차원 공간의  $N$ 개의 점들,  $\mathbf{p}_i$ 는  $pts$ 에

$$\mathbf{p}_0(1), \mathbf{p}_0(2), \dots, \mathbf{p}_0(m), \dots, \mathbf{p}_{N-1}(1), \dots, \mathbf{p}_{N-1}(m)$$

의 순서대로 저장되며, 최종적으로 이 method는 이것을 **vector<point>** 타입의 객체로 만들어 반환한다.

⟨Evaluation and derivative of **cubic\_spline 110**⟩ +≡

```

1490     vector<point>
1491     cubic_spline::evaluate_all(const unsigned N) const {
1492         const unsigned n = 3;
1493         const unsigned L = static_cast<unsigned>(_knot_sqnc.size() - 2 * n + 1);
1494         const unsigned m = static_cast<unsigned>(this->dim());
1495         size_t pts_buffer = _mp.create_buffer(mpoi::buffer_property::READ_WRITE, N * m * sizeof(float));
1496         ⟨Calculate points on a cubic spline using OpenCL Kernel 114⟩;
1497         float *pts = new float[N * m];
1498         _mp.enqueue_read_buffer(pts_buffer, N * m * sizeof(float), pts);
1499         vector<point> crv(N, point(m));
1500         for (size_t i = 0; i < N; i++) {
1501             point pt(m);
1502             for (size_t j = 1; j < m + 1; j++) {
1503                 pt(j) = static_cast<double>(pts[m * i + j - 1]);
1504             }
1505             crv[i] = pt;
1506         }
1507         delete[] pts;
1508         _mp.release_buffer(pts_buffer);
1509         return crv;
1510     }

```

**113.** ⟨Methods of **cubic\_spline 103**⟩ +≡

```

1511     public:
1512         vector<point> evaluate_all(const unsigned) const;

```



114. OpenCL로 작성한 kernel을 이용해서 spline 곡선상의 점들을 한꺼번에 계산한다. Kernel에서 de Boor 알고리즘을 계산하려면

1. knot sequence and its cardinality
2. control points and their cardinality
3. evaluation abscissa
4. evaluation abscissa가 속한 knot sequence 구간의 index

를 모두 넘겨줘야한다. 첫 번째와 두 번째는 kernel의 모든 work item들이 공유하지만, 세 번째와 네 번째는 work item마다 자신의 고유한 값을 갖고 연산을 수행한다.

가장 먼저 수행할 작업은 곡선의 knot sequence와 control points를 표준 라이브러리의 **vector** 타입으로부터 꺼내 단일한 memory block으로 복사하는 일이다. Knot sequence는 scalar 값이므로 순서대로 복사하고,  $m$  차원 공간의 control point들도 순서대로 모든 원소들을 복사한다.  $k$ 개의 control point들,  $\mathbf{d}_i$ 가 있다면 하나의 **double**형 배열에 아래와 같이 저장된다:

$$\mathbf{d}_1(1), \mathbf{d}_1(2), \dots, \mathbf{d}_1(m), \dots, \mathbf{d}_k(1), \dots, \mathbf{d}_k(m)$$

그 다음 OpenCL device의 memory에 입출력 data를 저장할 buffer object을 생성하고, memory buffer에 입력 data를 복사한다. (OpenCL kernel은 항상 **void**를 반환한다.)

〈Calculate points on a cubic spline using OpenCL Kernel 114〉 ≡

```

1513   const unsigned num_knots = static_cast<unsigned>(_knot_sqnc.size());
1514   const unsigned num_ctrlpts = static_cast<unsigned>(_ctrl_pts.size());
1515   float *knots = new float[num_knots];
1516   float *cp = new float[num_ctrlpts * m];
1517   size_t knots_buffer = _mp.create_buffer(mpoi::buffer_property::READ_ONLY,
      num_knots * sizeof(float));
1518   size_t cp_buffer = _mp.create_buffer(mpoi::buffer_property::READ_ONLY,
      num_ctrlpts * m * sizeof(float));
1519   for (size_t i = 0; i < num_knots; i++) {
1520       knots[i] = static_cast<float>(_knot_sqnc[i]);
1521   }
1522   for (size_t i = 0; i < num_ctrlpts; i++) {
1523       for (size_t j = 0; j < m; j++) {
1524           cp[i * m + j] = static_cast<float>(_ctrl_pts[i](j + 1));
1525       }
1526   }
1527   _mp.enqueue_write_buffer(knots_buffer, num_knots * sizeof(float), knots);
1528   _mp.enqueue_write_buffer(cp_buffer, num_ctrlpts * m * sizeof(float), cp);
1529   delete[] knots;
1530   delete[] cp;
1531   _mp.set_kernel_argument(_kernel_id, 0, pts_buffer);
1532   _mp.set_kernel_argument(_kernel_id, 1, knots_buffer);
1533   _mp.set_kernel_argument(_kernel_id, 2, cp_buffer);

```

```
1534     _mp.set_kernel_argument(_kernel_id, 3, sizeof(unsigned), (void *) &m);  
1535     _mp.set_kernel_argument(_kernel_id, 4, sizeof(unsigned), (void *) &L);  
1536     _mp.set_kernel_argument(_kernel_id, 5, sizeof(unsigned), (void *) &N);  
1537     _mp.enqueue_data_parallel_kernel(_kernel_id, N, 40);
```

이 코드는 112번 마디에서 사용된다.

**115.** 곡선을 계산하는 de Boor 알고리즘의 OpenCL 구현. Work item별로 따로 사용하는 private memory는 동적 할당을 지원하지 않는다. 따라서 부득이하게 고정된 크기의 배열을 사용하는데, cubic spline이므로  $n$ 은 3으로 고정하고, 허용하는 곡선의 최고 차원은 6으로 설정했다. 이는 OpenCL device의 spec에 따라 더 높이는 것이 가능하다.

이 함수에서 가장 먼저 수행할 작업은 domain knots,  $[u_{n-1}, \dots, u_{L+n-1}]$ 을  $N-1$ 개의 등간격으로 각 work item별로 자신이 계산해야 할  $u$  값을 계산하고,  $u$ 에 대하여  $u_i \in [u_I, u_{I+1}]$ 을 만족하는  $I$  값을 계산한다.

OpenCL 디바이스는 계산 유닛(Compute Unit)들로 이루어지고, 계산 유닛은 한 개 이상의 PE (Processing Element)들로 이루어진다. 디바이스에서의 실제 계산은 PE 안에서 이루어진다. 다수의 PE들이 같은 명령어를 실행한다는 점(SIMT; Single Instruction, Multiple Threads)을 생각해보면, kernel program 안에 분기문이 들어 있을 때 계산 성능이 저하된다. 따라서  $I$ 를 계산하는 과정에서 필요한 if 문을 그것과 동일한 효과를 내는 연산식으로 대체했음에 유의한다.

`<cspline.cl 115> ≡`

```

1538 #define MAX_BUFF_SIZE 30
1539 kernel void evaluate_crv(
1540     global float *crv,
1541     constant float *knots,
1542     constant float *cpts,
1543     unsigned d, unsigned L, unsigned N
1544 ) {
1545     private unsigned id = get_global_id(0);
1546     private const unsigned n = 3;
1547     private float tmp[MAX_BUFF_SIZE];
1548     private const float du = (knots[L + n - 1] - knots[n - 1]) / (float)(N - 1);
1549     private float u = knots[n - 1] + id * du;
1550     private unsigned I = n - 1;
1551     for (private unsigned i = n; i ≠ L + n - 1; i++) {
1552         I += (convert_int(sign(u - knots[i])) + 1) >> 1; /* If knots[i] < u, increment I. */
1553     }
1554     for (private unsigned i = 0; i ≠ n + 1; i++) {
1555         for (private unsigned j = 0; j ≠ d; j++) {
1556             tmp[i * d + j] = cpts[(i + I - n + 1) * d + j];
1557         }
1558     }
1559     private unsigned shifter = I - n + 1;
1560     for (private unsigned k = 1; k ≠ n + 1; k++) {
1561         for (private unsigned i = I + 1; i ≠ I - n + k; i--) {
1562             private float t = (knots[i + n - k] - u) / (knots[i + n - k] - knots[i - 1]);
1563             for (private unsigned j = 0; j ≠ d; j++) {
1564                 tmp[(i - shifter) * d + j] = t * tmp[(i - shifter - 1) * d + j] + (1. - t) * tmp[(i - shifter) * d + j];
1565             }
1566         }

```

```

1567     }
1568     for (private unsigned j = 0; j ≠ d; j++) {
1569         crv[id * d + j] = tmp[n * d + j];
1570     }
1571 }

```

**116.** 어떤 scalar 값이 주어졌을 때, 그것이 knot sequence의 몇 번째 knot과 그 다음 knot 사이에 들어가는 값인지 찾아내는 method를 정의한다. 즉,  $u$ 가 주어지면,  $u_i \leq u < u_{i+1}$ 을 만족하는 인덱스  $i$ 를 찾는 것이다. 만약 조건을 만족하는  $i$ 가 없으면, `SIZE_MAX`를 반환한다. 이 method는 non-decreasing knot sequence를 가정하며, 만약  $u$ 가 knot sequence의 마지막 값과 같다면 조건식이 만족되지 않으므로 sequence를 뒤에서부터 거슬러  $u_i \leq u \leq u_{i+1}$ 을 만족하는  $i$ 를 찾는다. 이는 knot의 multiplicity가 2 이상일 때에도 대응하기 위함이다. 이 method는 하나의 `double` 타입 인자만 주어지면 객체의 knot sequence에서 해당하는 인덱스를 찾지만, 별도의 knot sequence가 주어지면 주어진 sequence에서 인덱스를 찾는다.

⟨Miscellaneous methods of `cubic_spline` 116⟩ ≡

```

1572     size_t
1573     cubic_spline::find_index_in_sequence(
1574         const double u,
1575         const vector<double> sqnc
1576     ) const {
1577         if (u ≡ sqnc.back()) {
1578             for (size_t i = sqnc.size() - 2; i ≠ SIZE_MAX; i--) {
1579                 if (sqnc[i] ≠ u) {
1580                     return i;
1581                 }
1582             }
1583         }
1584         for (size_t i = 0; i ≠ sqnc.size() - 1; i++) {
1585             if ((sqnc[i] ≤ u) ∧ (u < sqnc[i + 1])) {
1586                 return i;
1587             }
1588         }
1589         return SIZE_MAX;
1590     }
1591     size_t
1592     cubic_spline::find_index_in_knot_sequence(const double u) const {
1593         return find_index_in_sequence(u, this->knot_sqnc);
1594     }

```

122, 124, 126, 128, 182번 마디도 살펴보자.

이 코드는 101번 마디에서 사용된다.

117. 〈Methods of **cubic\_spline** 103〉 +≡1595 **protected:**

```

1596     size_t find_index_in_sequence(
1597         const double,
1598         const vector<double>
1599     ) const;
1600     size_t find_index_in_knot_sequence(const double) const;
```

118. Spline 곡선의 미분은 동등한 Bézier 곡선으로 변환한 후 계산한다.

〈Evaluation and derivative of **cubic\_spline** 110〉 +≡

```

1601     point cubic_spline::derivative(const double u) const {
1602         〈Check the range of knot value given 120〉;
1603         vector<point> splines, bezier_ctrlpt;
1604         vector<double> knots;
1605         bezier_control_points(splines, knots);    /* Equivalent Bézier curves. */
1606         unsigned long index = find_index_in_sequence(u, knots);
1607         for (size_t i = index * 3; i ≤ (index + 1) * 3; i++) {
1608             bezier_ctrlpt.push_back(splines.at(i));
1609         }
1610         bezier bezier_curve = bezier(bezier_ctrlpt);
1611         double delta = knots[index + 1] - knots[index];
1612         double t = (u - knots[index]) / delta;    /* Change coordinate from b-spline to Bézier. */
1613         point drv(bezier_curve.derivative(t));
1614         return drv / delta;    /* Transform the velocity into the u coordinate (b-spline). */
1615     }
```

119. 〈Methods of **cubic\_spline** 103〉 +≡1616 **public:**1617 **point** derivative(**const double**) **const**;120. 만약 주어진 인자  $u$ 가 knot sequence의 범위를 벗어나면, OUT\_OF\_KNOT\_RANGE 오류코드를 객체에 남기고 모든 원소가 0인 **point** 객체를 반환한다. 객체의 차원은 컨트롤 포인트의 차원과 동일하다.

〈Check the range of knot value given 120〉 ≡

```

1618     if ((u < _knot_sqnc.front()) ∨ (_knot_sqnc.back() < u)) {
1619         _err = OUT_OF_KNOT_RANGE;
1620         return cagd::point(_ctrl_pts.begin()-dim());
1621     }
```

이 코드는 118번 마디에서 사용된다.

121. 〈Error codes of **cagd** 34〉 +≡

1622 OUT\_OF\_KNOT\_RANGE ,

**122.** Knot의 multiplicity를 찾는 method는 재귀적으로 구현한다. 즉, sequence의 시작점부터 주어진 knot과 같은 knot을 찾을때마다 다시 같은 함수를 호출한다.

⟨ Miscellaneous methods of **cubic\_spline** 116 ⟩ +≡

```

1623   unsigned long cubic_spline::find_multiplicity(const double u, const_knot_itr begin) const {
1624       const_knot_itr iter = find(begin, _knot_sqnc.end(), u);
1625       if (iter == _knot_sqnc.end()) {
1626           return 0;
1627       } else {
1628           return find_multiplicity(u, ++iter) + 1;
1629       }
1630   }
1631   unsigned long cubic_spline::find_multiplicity(const double u) const {
1632       return find_multiplicity(u, _knot_sqnc.begin());
1633   }

```

**123.** ⟨ Methods of **cubic\_spline** 103 ⟩ +≡

```

1634   protected:
1635       unsigned long find_multiplicity(const double, const_knot_itr) const;
1636       unsigned long find_multiplicity(const double) const;

```

**124.** Knot sequence의 증분값,  $\Delta u_i$ 를 계산하는 간단한 method를 정의한다. 이 프로그램의 많은 부분에서  $\Delta_i = \Delta u_i = u_{i+1} - u_i$ 를 의미하며, 편의상  $\Delta_{-1} = \Delta_L = 0$ 을 반환하도록 구현한다. 이는 보간 (interpolation) 방정식의 구현을 간단하게 만들어준다.

⟨ Miscellaneous methods of **cubic\_spline** 116 ⟩ +≡

```

1637   double cubic_spline::delta(const long i) const {
1638       if ((i < 0) ∨ (_knot_sqnc.size() - 1) ≤ i) {
1639           return 0.;
1640       } else {
1641           return _knot_sqnc[i + 1] - _knot_sqnc[i];
1642       }
1643   }

```

**125.** ⟨ Methods of **cubic\_spline** 103 ⟩ +≡

```

1644   protected:
1645       double delta(const long) const;

```

**126.** Knot sequence의 양 끝에 곡선의 차수만큼 knot을 추가해서 곡선이 양 끝의 컨트롤 포인트를 지나도록 하는 method를 정의한다.

⟨ Miscellaneous methods of **cubic\_spline** 116 ⟩ +=

```

1646     void cubic_spline::insert_end_knots()
1647     {
1648         vector<double> newKnots;
1649         newKnots.push_back(_knot_sqnc[0]);
1650         newKnots.push_back(_knot_sqnc[0]);
1651         for (size_t i = 0; i ≠ _knot_sqnc.size(); ++i) {
1652             newKnots.push_back(_knot_sqnc[i]);
1653         }
1654         newKnots.push_back(_knot_sqnc.back());
1655         newKnots.push_back(_knot_sqnc.back());
1656         _knot_sqnc.clear();
1657         for (size_t i = 0; i ≠ newKnots.size(); ++i) {
1658             _knot_sqnc.push_back(newKnots[i]);
1659         }
1660     }

```

**127.** ⟨ Methods of **cubic\_spline** 103 ⟩ +=

```

1661     protected:
1662     void insert_end_knots();

```

**128.** Cubic spline 곡선의 control point들을 주어진 point들로 대체하는 method. 이는 주로 cubic spline interpolation의 계산 결과를 반영하는 것을 염두에 두고 있어서, 양 끝점들과 중간 점들의 **vector** 타입을 입력으로 받는다.

⟨ Miscellaneous methods of **cubic\_spline** 116 ⟩ +=

```

1663     void cubic_spline::set_control_points(
1664         const point &head,
1665         const vector<point> &intermediate,
1666         const point &tail
1667     ) {
1668         _ctrl_pts.clear();
1669         size_t n = intermediate.size();
1670         _ctrl_pts = vector<point>(2 + n, point(2));
1671         _ctrl_pts[0] = head;
1672         for (size_t i = 0; i ≠ n; i++) {
1673             _ctrl_pts[i + 1] = intermediate[i];
1674         }
1675         _ctrl_pts[n + 1] = tail;
1676     }

```

**129.**  $\langle$  Methods of **cubic\_spline** 103  $\rangle + \equiv$

1677 **protected:**

1678 **void** *set\_control\_points*(**const** **point** &, **const** **vector** $\langle$ **point** $\rangle$  &, **const** **point** &);



**130.** Inversion of a tridiagonal matrix.

Cubic spline 곡선의 보간법을 다루기 위하여 먼저 tridiagonal system의 해법을 설명하고 구현한다.

Riaz A. Usmani, “Inversion of a Tridiagonal Jacobi Matrix,” *Linear Algebra and its Applications*, **212**, 1994, pp. 413–414와 C. M. da Fonseca, “On the Eigenvalues of Some Tridiagonal Matrices,” *J. Computational and Applied Mathematics*, **200**(1), 2007, pp. 283–286을 참고하면 tridiagonal matrix의 역행렬은 간단한 계산으로 구할 수 있다.

행렬

$$T = \begin{pmatrix} \beta_1 & \gamma_1 & & & \\ \alpha_2 & \beta_2 & \gamma_2 & & \\ & & \ddots & & \\ & & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ & & & & \alpha_n & \beta_n \end{pmatrix}$$

의 역행렬  $T^{-1}$ 의 원소는 다음과 같이 주어진다.

$$(T^{-1})_{ij} = \begin{cases} (-1)^{i+j} \gamma_i \cdots \gamma_{j-1} \theta_{i-1} \phi_j / \theta_n, & \text{if } i < j; \\ \theta_{i-1} \phi_j / \theta_n, & \text{if } i = j; \\ (-1)^{i+j} \alpha_j \cdots \alpha_{i-1} \theta_{j-1} \phi_i / \theta_n, & \text{if } i > j. \end{cases}$$

이 때  $\theta_i$ 와  $\phi_i$ 는 다음의 점화식으로부터 얻는다.

$$\begin{aligned} \theta_i &= \beta_i \theta_{i-1} - \gamma_{i-1} \alpha_{i-1} \theta_{i-2} & (i = 2, 3, \dots, n), \\ \phi_i &= \beta_{i+1} \phi_{i+1} - \gamma_{i+1} \alpha_{i+1} \phi_{i+2} & (i = n-2, \dots, 0). \end{aligned}$$

이 점화식들의 초기 조건은

$$\begin{aligned} \theta_0 &= 1, \quad \theta_1 = \beta_1; \\ \phi_{n-1} &= \beta_n, \quad \phi_n = 1 \end{aligned}$$

이다.

정리하면, tridiagonal matrix의 역행렬을 구하는 과정은 다음과 같다:

1.  $\theta_0$ 와  $\theta_1$ 을 이용하여  $\theta_2, \dots, \theta_n$ 을 계산;
2.  $\theta_n = 0$ 이면 행렬이 비가역이므로 계산 종료. 그렇지 않으면 나머지 단계로 진행;
3.  $\phi_{n+1}$ 과  $\phi_n$ 을 이용하여  $\phi_{n-1}, \dots, \phi_1$ 을 계산;
4.  $\phi_i$ 와  $\theta_i$ 들을 이용하여 역행렬의 원소들을 계산.

여기에서 정의하는 `invert_tridiagonal()` 함수는 계산한 역행렬을 row-major order, 즉 첫 번째 행부터 마지막 행까지 하나의 **vector**에 순서대로 넣어 반환한다. 행렬이 비가역적이면 함수는  $-1$ 을, 가역이면  $0$ 을 반환한다.

〈Implementation of **cagd** functions 4〉 +≡

```

1679  int cagd::invert_tridiagonal(
1680      const vector<double> &alpha,
1681      const vector<double> &beta,
1682      const vector<double> &gamma,
1683      vector<double> &inverse
1684  ) {
1685      size_t n = beta.size();
1686      vector<double> theta(n+1, 0.);    /* From 0 to n. */
1687      theta[0] = 1.;

```

```

1688  theta[1] = beta[0];
1689  for (size_t i = 2; i ≠ n + 1; i++) {
1690      theta[i] = beta[i - 1] * theta[i - 1] - gamma[i - 2] * alpha[i - 2] * theta[i - 2];
1691  }
1692  if (theta[n] ≡ 0.) return -1;    /* The matrix is singular. */
1693  vector<double> phi(n + 1, 0.);    /* From 0 to n. */
1694  phi[n] = 1.;
1695  phi[n - 1] = beta[n - 1];
1696  for (size_t i = n - 1; i ≠ 0; i--) {
1697      phi[i - 1] = beta[i - 1] * phi[i] - gamma[i - 1] * alpha[i - 1] * phi[i + 1];
1698  }
1699  for (size_t i = 0; i ≠ n; i++) {
1700      for (size_t j = 0; j ≠ n; j++) {
1701          double elem = 0.;
1702          if (i < j) {
1703              double prod = 1.;
1704              for (size_t k = i; k ≠ j; k++) {
1705                  prod *= gamma[k];
1706              }
1707              elem = pow(-1, i + j) * prod * theta[i] * phi[j + 1] / theta[n];
1708          }
1709          else if (i ≡ j) {
1710              elem = theta[i] * phi[j + 1] / theta[n];
1711          }
1712          else {
1713              double prod = 1.;
1714              for (size_t k = j; k ≠ i; k++) {
1715                  prod *= alpha[k];
1716              }
1717              elem = pow(-1, i + j) * prod * theta[j] * phi[i + 1] / theta[n];
1718          }
1719          inverse[i * n + j] = elem;
1720      }
1721  }
1722  return 0;    /* No error. */
1723  }

```

**131.** 〈Declaration of **cagd** functions 5〉 +≡

```

1724   int invert_tridiagonal(
1725       const vector<double> &,
1726       const vector<double> &,
1727       const vector<double> &,
1728       vector<double> &);

```

**132.** Test: Inversion of a Tridiagonal Matrix.

예제로

$$\begin{pmatrix} 1 & 4 & 0 & 0 \\ 3 & 4 & 1 & 0 \\ 0 & 2 & 3 & 4 \\ 0 & 0 & 1 & 3 \end{pmatrix}$$

의 역행렬을 계산한다. 결과는

$$\begin{pmatrix} -0.304348 & 0.434783 & -0.26087 & 0.347826 \\ 0.326087 & -0.108696 & 0.0652174 & -0.0869565 \\ -0.391304 & 0.130435 & 0.521739 & -0.695652 \\ 0.130435 & -0.0434783 & -0.173913 & 0.565217 \end{pmatrix}$$

이다.

## 〈Test routines 26〉 +≡

```

1729   print_title("inversion_of_a_tridiagonal_matrix");
1730   {
1731       vector<double> alpha(3,0.);
1732       alpha[0] = 3.; alpha[1] = 2.; alpha[2] = 1.;
1733       vector<double> beta(4,0.);
1734       beta[0] = 1.; beta[1] = 4.; beta[2] = 3.; beta[3] = 3.;
1735       vector<double> gamma(3,0.);
1736       gamma[0] = 4.; gamma[1] = 1.; gamma[2] = 4.;
1737       vector<double> inv(4 * 4, 0.);
1738       cagd::invert_tridiagonal(alpha, beta, gamma, inv);
1739       for (size_t i = 0; i < 4; i++) {
1740           for (size_t j = 0; j < 4; j++) {
1741               cout << inv[i * 4 + j] << " ";
1742           }
1743           cout << endl;
1744       }
1745   }

```

**133.** Multiplication of a matrix and a vector. Tridiagonal matrix의 역행렬을 이용하여 tridiagonal system의 해를 구하려면, 일반적인 행렬과 벡터의 곱셈이 필요하다. 여기서는 row-major order로 하나의 **vector** 타입 객체에 저장된 정방행렬과 하나의 **vector** 타입 객체에 저장되어 있는 column vector의 곱셈을 구현한다.

〈Implementation of **cagd** functions 4〉 +≡

```

1746     vector<double> cagd::multiply(
1747         const vector<double> &mat,
1748         const vector<double> &vec
1749     ) {
1750         size_t n = vec.size();
1751         vector<double> mv(n, 0.);
1752         for (size_t i = 0; i < n; i++) {
1753             for (size_t k = 0; k < n; k++) {
1754                 mv[i] += mat[i * n + k] * vec[k];
1755             }
1756         }
1757         return mv;
1758     }

```

**134.** 〈Declaration of **cagd** functions 5〉 +≡

```

1759     vector<double> multiply(
1760         const vector<double> &,
1761         const vector<double> &);

```

**135.** Tridiagonal matrix의 역행렬을 이용하여 tridiagonal system의 해를 구하는 것은 매우 간단하다.

$$Ax = b$$

에서 세 개의 **vector**(**double**) 타입의 입력인자,  $l$ ,  $d$ ,  $u$ 는 각각  $n \times n$  행렬  $A$ 의 lower diagonal, diagonal, upper diagonal element들이다.  $l$ 과  $u$ 는  $n - 1$ 개,  $d$ 는  $n$ 개의 원소를 가져야 한다. **vector**(**point**) 타입의 인자  $b$ 와  $x$ 는 각각 방정식의 우변과 해를 의미한다. 방정식의 해가 유일하게 존재하면 함수는 0을, 그렇지 않으면  $-1$ 을 반환한다.

〈Implementation of **cagd** functions 4〉 +≡

```

1762     int cagd :: solve_tridiagonal_system(
1763         const vector<double> &l,
1764         const vector<double> &d,
1765         const vector<double> &u,
1766         const vector<point> &b,
1767         vector<point> &x
1768     ) {
1769         size_t n = d.size();
1770         vector<double> Ainv(n * n, 0.);
1771         if (cagd :: invert_tridiagonal(l, d, u, Ainv) ≠ 0) return -1;
1772         for (size_t i = 1; i ≠ b[0].dim() + 1; i++) {
1773             vector<double> r(n, 0.);
1774             for (size_t k = 0; k ≠ n; k++) {
1775                 r[k] = b[k](i);
1776             }
1777             vector<double> xi = cagd :: multiply(Ainv, r);
1778             for (size_t k = 0; k ≠ n; k++) {
1779                 x[k](i) = xi[k];
1780             }
1781         }
1782         return 0;
1783     }

```

**136.** 〈Declaration of **cagd** functions 5〉 +≡

```

1784     int solve_tridiagonal_system(
1785         const vector<double> &,
1786         const vector<double> &,
1787         const vector<double> &,
1788         const vector<point> &,
1789         vector<point> &);

```

**137.** Ahlberg-Nilson-Walsh Algorithm. (Solution of a cyclic tridiagonal system.)

Tridiagonal system을 구성하는 관계식이 시작점과 끝점에서도 꼬리에 꼬리를 무는 형태로 반복되는 경우 cyclic tridiagonal system이라 부르며, Ahlberg-Nilson-Walsh algorithm (Clive Temperton, “Algorithms for the Solution of Cyclic Tridiagonal Systems,” *J. Computational Physics*, **19**(3), 1975, pp. 317–323)을 참조하면 일반적인 linear system의 해법을 쓰지 않고 변형된 tridiagonal system으로 풀 수 있다.

방정식

$$\begin{pmatrix} \beta_1 & \gamma_1 & & & & \alpha_1 \\ \alpha_2 & \beta_2 & \gamma_2 & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ \gamma_n & & & & \alpha_n & \beta_n \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

이 주어졌을 때,

$$\left( \begin{array}{cccc|c} \beta_1 & \gamma_1 & & & \alpha_1 \\ \alpha_2 & \beta_2 & \gamma_2 & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ \hline & & & & \alpha_n & \beta_n & \gamma_n \end{array} \right) = \begin{pmatrix} E & f \\ g^\top & h \end{pmatrix}, \quad \begin{pmatrix} x_1 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{x}} \\ x_n \end{pmatrix}, \quad \begin{pmatrix} b_1 \\ \vdots \\ b_{n-1} \\ b_n \end{pmatrix} = \begin{pmatrix} \hat{\mathbf{b}} \\ b_n \end{pmatrix}$$

으로 치환하면,

$$E\hat{\mathbf{x}} + fx_n = \hat{\mathbf{b}}$$

$$g^\top \hat{\mathbf{x}} + hx_n = b_n$$

이고, tridiagonal matrix  $E$ 는 쉽게 역행렬을 구할 수 있으므로

$$\hat{\mathbf{x}} = E^{-1}(\hat{\mathbf{b}} - fx_n)$$

을 두 번째 방정식에 대입하면

$$x_n = \frac{b_n - g^\top E^{-1} \hat{\mathbf{b}}}{h - g^\top E^{-1} f}$$

이고,

$$\hat{\mathbf{x}} = E^{-1} \left( \hat{\mathbf{b}} - f \frac{b_n - g^\top E^{-1} \hat{\mathbf{b}}}{h - g^\top E^{-1} f} \right)$$

이다.

아래 함수는 입력 인자, *alpha*, *beta*, *gamma*가 각각  $\alpha_i$ ,  $\beta_i$ ,  $\gamma_i$ 들을 담고 있음을 가정한다.

⟨Implementation of **cagd** functions 4⟩ +≡

```

1790  int cagd::solve_cyclic_tridiagonal_system(
1791      const vector<double> &alpha,
1792      const vector<double> &beta,
1793      const vector<double> &gamma,
1794      const vector<point> &b,
1795      vector<point> &x
1796  ) {
1797      size_t n = beta.size();

```

```

1798     vector<double> Einv((n - 1) * (n - 1), 0.);
1799     < Calculate  $E^{-1}$  138 >;
1800     size_t dim = b[0].dim();
1801     vector<vector<double>> B(dim, vector<double>(n, 0.));
1802     for (size_t i = 0; i ≠ dim; i++) {
1803         for (size_t j = 0; j ≠ n; j++) {
1804             B[i][j] = b[j](i + 1);
1805         }
1806         < Calculate  $x_n$  139 >;
1807         < Calculate  $\hat{\mathbf{x}}$  140 >;
1808         for (size_t j = 0; j ≠ n - 1; j++) {
1809             x[j](i + 1) = xhat[j];
1810         }
1811         x[n - 1](i + 1) = x_n;
1812     }
1813     return 0;
1814 }
```

138. < Calculate  $E^{-1}$  138 > ≡

```

1815     vector<double> l = vector<double>(n - 2, 0.);
1816     vector<double> d = vector<double>(n - 1, 0.);
1817     vector<double> u = vector<double>(n - 2, 0.);
1818     for (size_t j = 0; j ≠ n - 2; j++) {
1819         l[j] = alpha[j + 1];
1820         d[j] = beta[j];
1821         u[j] = gamma[j];
1822     }
1823     d[n - 2] = beta[n - 2];
1824     if (invert_tridiagonal(l, d, u, Einv) ≠ 0) return -1;
```

이 코드는 137번 마디에서 사용된다.

139.  $g$ 와  $f$ 의 특성으로 인하여

$$\begin{aligned} g^\top E^{-1} f &= \gamma_n (\alpha_1 E_{1,1}^{-1} + \gamma_{n-1} E_{1,n-1}^{-1}) + \alpha_n (\alpha_1 E_{n-1,1}^{-1} + \gamma_{n-1} E_{n-1,n-1}^{-1}); \\ g^\top E^{-1} \hat{\mathbf{b}} &= \gamma_n (E_{1,1}^{-1} b_1 + \cdots + E_{1,n-1}^{-1} b_{n-1}) + \alpha_n (E_{n-1,1}^{-1} b_1 + \cdots + E_{n-1,n-1}^{-1} b_{n-1}) \end{aligned}$$

이다.

〈Calculate  $x_n$  139〉  $\equiv$

```
1825   double x_n_den = beta[n-1] - gamma[n-1] * (alpha[0] * Einv[0] + gamma[n-2] * Einv[n-2]) -
      alpha[n-1] * (alpha[0] * Einv[(n-2) * (n-1)] + gamma[n-2] * Einv[(n-1) * (n-1) - 1]);
1826   double E1b = 0.;
1827   double Enb = 0.;
1828   for (size_t j = 0; j < n-1; j++) {
1829       E1b += Einv[j] * B[i][j];
1830       Enb += Einv[(n-2) * (n-1) + j] * B[i][j];
1831   }
1832   double x_n_num = B[i][n-1] - gamma[n-1] * E1b - alpha[n-1] * Enb;
1833   double x_n = x_n_num / x_n_den;
```

이 코드는 137번 마디에서 사용된다.

140. 〈Calculate  $\hat{\mathbf{x}}$  140〉  $\equiv$

```
1834   vector<double> bhat_fx(n-1, 0.);
1835   for (size_t j = 0; j < n-1; j++) {
1836       bhat_fx[j] = B[i][j];
1837   }
1838   bhat_fx[0] -= alpha[0] * x_n;
1839   bhat_fx[n-2] -= gamma[n-2] * x_n;
1840   vector<double> xhat = multiply(Einv, bhat_fx);
```

이 코드는 137번 마디에서 사용된다.

141. 〈Declaration of **cagd** functions 5〉  $+\equiv$

```
1841   int solve_cyclic_tridiagonal_system(
1842       const vector<double> &,
1843       const vector<double> &,
1844       const vector<double> &,
1845       const vector<point> &,
1846       vector<point> &);
```



**142.** Test: Cyclic Tridiagonal System.

예제로

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 1 & 7 \\ 2 & 6 \\ 3 & 5 \\ 4 & 4 \\ 5 & 3 \\ 6 & 2 \\ 7 & 1 \end{pmatrix}$$

일 때,  $A\mathbf{x} = \mathbf{b}$ 의 해를 구하면,

$$\mathbf{x} = \begin{pmatrix} -5 & 7 \\ 4 & -2 \\ -1 & 3 \\ 1 & 1 \\ 3 & -1 \\ -2 & 4 \\ 7 & -5 \end{pmatrix}$$

이다.

〈Test routines 26〉 +≡

```

1847   print_title("cyclic_tridiagonal_system");
1848   {
1849       vector<double> alpha(7, 1.);
1850       vector<double> beta(7, 2.);
1851       vector<double> gamma(7, 1.);
1852       vector<point> b(7, point(2));
1853       b[0] = point({1., 7.});
1854       b[1] = point({2., 6.});
1855       b[2] = point({3., 5.});
1856       b[3] = point({4., 4.});
1857       b[4] = point({5., 3.});
1858       b[5] = point({6., 2.});
1859       b[6] = point({7., 1.});
1860       vector<point> x(7, point(2));
1861       solve_cyclic_tridiagonal_system(alpha, beta, gamma, b, x);
1862       cout << "x_=" << endl;
1863       for (size_t i = 0; i < 7; i++) {
1864           cout << "[" << x[i](1) << ", " << x[i](2) << "]" << endl;
1865       }
1866   }

```

**143.** `cubic_spline`의 보간법에서 사용하기 위한 상수들을 enumeration으로 정의한다. `parametrization`은 곡선의 knot sequence를 어떻게 생성할 것인지를 기술하기 위한 상수들이다. 각각 uniform parametrization, chord length parametrization, centripetal parametrization, spline function parametrization을 의미한다.

`end_condition`은 곡선의 end condition을 어떻게 설정할 것인지 나타낸다. 각각 clamped, Bessel, quadratic, not-a-knot, natural, 그리고 끝으로 periodic end condition을 의미한다.

〈Enumerations of `cubic_spline` 143〉 ≡

```

1867 public:
1868     enum class parametrization {
1869         uniform,      /* uniform parametrization */
1870         chord_length,  /* chord length parametrization */
1871         centripetal,   /* centripetal parametrization */
1872         function_spline /* spline function, i.e., knot sequence = x coords. */
1873     };
1874     enum class end_condition {
1875         clamped,      /* claped end condition */
1876         bessel,       /* Bessel end condition */
1877         quadratic,    /* quadratic end condition */
1878         not_a_knot,   /* not-a-knot end condition */
1879         natural,      /* natural end condition */
1880         periodic      /* periodic end condition */
1881     };

```

이 코드는 99번 마디에서 사용된다.

**144.** Cubic spline 보간은 데이터 포인트,  $\mathbf{p}_0, \dots, \mathbf{p}_L$ 이 주어져 있을 때, 그 데이터 포인트들을 지나면서  $C^2$  연속성 조건을 만족하는 spline curve의 컨트롤 포인트,  $\mathbf{d}_{-1}, \dots, \mathbf{d}_{L+1}$ 를 찾는 것이다. 주어진 데이터 포인트는  $L + 1$ 개이고, 찾아야하는 컨트롤 포인트는  $L + 3$ 개이므로 이는 부정방정식(under-determined problem)이다. 따라서 문제의 유일해를 구하려면 2개의 구속조건이 더 주어져야하며, 이는 end-condition에 의하여 결정한다.

엄밀하게 말하면, cubic spline 보간은 데이터 포인트  $\mathbf{p}_0, \dots, \mathbf{p}_L$  뿐만 아니라 knot sequence,  $u_0, \dots, u_L$ , 그리고 각 knot들의 multiplicity가 주어져야 해를 구할 수 있다. 그러나 일반적으로 knot sequence와 multiplicity는 주어지지 않으므로 knot sequence는 몇 가지 scheme을 선택하도록 해서 그에 따라 생성하고, knot들의 multiplicity는 곡선이 양 끝점의 data point를 지나갈 수 있도록 3, 1,  $\dots$ , 1, 3을 가정한다.

가장 먼저, 데이터 포인트, 매개화 (parametrization) scheme, 종단 조건 (end condition), 종단 하나 이전의 컨트롤 포인트 ( $\mathbf{d}_0$ 와  $\mathbf{d}_L$ )을 모두 입력으로 받는 일반적인 보간 기능을 *interpolate()* 메소드로 구현한다. 이것은 모든 종류의 보간 문제를 해결하는 engine이다.

*interpolate()* 메소드는 주어진 데이터 포인트의 갯수가 0이면 knot sequence와 control point를 모두 비워 버린 후 바로 반환한다. 데이터 포인트의 갯수가 1이면 trivial solution으로 그 데이터 포인트를 유일한 컨트롤 포인트로, knot sequence는 0을 3개 중첩한 후 반환한다. 그렇지 않을 경우에는 주어진 parametrization scheme 따라 knot sequence를 생성하고,  $C^2$  cubic spline 보간에 관한 연립방정식을 세운 후, end condition에 맞춰 일부 식을 조작한다. 방정식의 해를 구함으로써 control point들을 구하고, 마지막으로 곡선 양 끝의 knot을 3개 중첩시키면 보간이 끝난다.

사용 편의성을 위해 경우에 따라 몇 가지 불필요한 인자들을 생략한 *interpolate()* 메소드들을 정의한다:

1. 데이터 포인트만 주어지거나, 데이터 포인트와 매개화 scheme이 함께 주어지면 not-a-knot 종단 조건을 가정한다. 매개화 scheme이 주어지지 않을 때에는 가장 범용적인 chord length 매개화를 가정한다. 데이터 포인트가 3점 이상 주어지는 경우, 양 끝에서 하나 이전의 컨트롤 포인트들은 not-a-knot 종단 조건에 의하여 결정되므로 큰 의미는 없지만, 데이터 포인트가 2점 주어지는 경우 그 두 점을 잇는 직선이 얻어질 수 있도록 양 끝의 데이터 포인트를 각각 1/3과 2/3로 내분하는 점을 계산한 후 engine method에 넘겨준다.
2. 데이터 포인트와 추가로 두 개의 포인트가 주어지면 clamped end condition을 가정한다. 매개화 scheme은 주어진 것을 사용하거나, 아니면 chord length 매개화를 가정한다.

(Methods for interpolation of **cubic\_spline** 144)  $\equiv$

```

1882 void
1883 cubic_spline::interpolate(const vector<point> &p, parametrization scheme, end_condition
      cond, const point &initial, const point &end) {
1884     _knot_sqnc.clear();
1885     _ctrl_pts.clear();
1886     if (p.size() == 0) { /* No data point given. */
1887     }
1888     else if (p.size() == 1) { /* A single data point. Trivial. */
1889         _knot_sqnc.push_back(0.);
1890         _knot_sqnc.push_back(0.);
1891         _knot_sqnc.push_back(0.);
1892         _ctrl_pts.push_back(p[0]);
1893         _ctrl_pts.push_back(p[0]);
1894         _ctrl_pts.push_back(p[0]);
1895     }
1896     else { /* More than or equal to 2 points given. */

```

```

1897     <Generate knot sequence according to given parametrization scheme 148>;
1898     <Setup equations of cubic spline interpolation 154>;
1899     <Modify equations according to end conditions and solve them 155>;
1900     insert_end_knots();
1901 }
1902 }

```

이 코드는 101번 마디에서 사용된다.

#### 145. <Methods of **cubic\_spline** 103> +≡

```

1903 protected:
1904     void _interpolate(const vector<point> &p, parametrization, end_condition, const point &i, const
        point &e);

```

146. 한편, 데이터 포인트들이 주어졌을 때 그것들을 보간하는 cubic spline 곡선을 바로 생성하는 constructor 가 있으면 매우 유용할 것이다. 아무런 parametrization scheme이나 end condition이 주어지지 않으면 chord length parametrization과 not-a-knot end condition을 적용한다.

#### <Constructors and destructor of **cubic\_spline** 102> +≡

```

1905     cubic_spline::cubic_spline(const vector<point> &p, end_condition cond, parametrization
        scheme)
1906         : curve(p),
1907           _mp("./cspline.cl"),
1908           _kernel_id(_mp.create_kernel("evaluate_crv"))
1909     {
1910         point one_third(2./3. * (*p.begin()) + 1./3. * (p.back()));
1911         point two_third(1./3. * (*p.begin()) + 2./3. * (p.back()));
1912         _interpolate(p, scheme, cond, one_third, two_third);
1913     }
1914     cubic_spline::cubic_spline(const vector<point> &p, const point i, const point e, parametrization
        scheme)
1915         : curve(p),
1916           _mp("./cspline.cl"),
1917           _kernel_id(_mp.create_kernel("evaluate_crv"))
1918     {
1919         _interpolate(p, scheme, end_condition::clamped, i, e);
1920     }

```

#### 147. <Methods of **cubic\_spline** 103> +≡

```

1921 public:
1922     cubic_spline(const vector<point> &p, end_condition cond = end_condition::not_a_knot,
        parametrization scheme = parametrization::chord_length);
1923     cubic_spline(const vector<point> &p, const point, const point, parametrization
        scheme = parametrization::chord_length);

```

**148.** 먼저 parametrization scheme에 따라 knot sequence를 적절하게 배치해야한다. **cubic\_spline** 타입은 uniform, chord length, centripetal, function spline parametrization을 지원한다. 알려지지 않은 scheme으로 parametrization을 시도하면 UNKNOWN\_PARAMETRIZATION 오류 코드를 객체 내에 저장하고 반환한다. 보통은 chord length parametrization이나 centripetal parametrization을 사용한다.

〈Generate knot sequence according to given parametrization scheme 148〉≡

```

1924  switch (scheme) {
1925  case parametrization::uniform: {
1926      〈Uniform parametrization of knot sequence 150〉;
1927  }
1928  break;
1929  case parametrization::chord_length: {
1930      〈Chord length parametrization of knot sequence 151〉;
1931  }
1932  break;
1933  case parametrization::centripetal: {
1934      〈Centripetal parametrization of knot sequence 152〉;
1935  }
1936  break;
1937  case parametrization::function_spline: {
1938      〈Function spline parametrization of knot sequence 153〉;
1939  }
1940  break;
1941  default:
1942      _err = UNKNOWN_PARAMETRIZATION;
1943      return;
1944  }

```

이 코드는 144번 마디에서 사용된다.

**149.** 〈Error codes of cagd 34〉 +≡

```

1945  UNKNOWN_PARAMETRIZATION ,

```

**150.** Uniform parametrization: 등간격으로 knot들을 배치한다. Data point의 갯수가  $L$  이라면,  $i$  번째 knot  $u_i = L$ 로 설정한다. 이는 data point들 사이의 거리를 고려하지 않기 때문에 point들이 촘촘한 구간에서는 곡선이 천천히, 멀리 떨어진 구간에서는 너무 빨리 움직이는 문제가 있어서 data point들 사이의 간격들이 균일하지 못하면 곡선의 품질면에서 불리한 knot sequence를 생성하게 된다.

〈Uniform parametrization of knot sequence 150〉≡

```

1946  for (size_t i = 0; i < p.size(); i++) {
1947      _knot_sqnc.push_back(double(i));
1948  }

```

이 코드는 148번 마디에서 사용된다.

**151.** Chord length parametrization: data point,  $x_i$ 들 사이의 거리(chord length)에 비례하여 knot들을 배치한다. 즉,  $u_{i+1} - u_i = \Delta_i$ 이고  $\|x_{i+1} - x_i\| = \Delta x_i$ 이면,

$$\frac{\Delta_i}{\Delta_{i+1}} = \frac{\|\Delta x_i\|}{\|\Delta x_{i+1}\|}$$

이 되도록 한다. 실제 구현에서는  $u_0 = 0$ 이고  $u_L = 1$ 이 되도록 하거나, 또는  $u_0 = 0$ 이고  $u_L = L$ 이 되도록 하는 것이 바람직하다.

〈Chord length parametrization of knot sequence 151〉≡

```

1949  _knot_sqnc.push_back(0.);    /* u_0 */
1950  double sum_delta = 0.;
1951  for (size_t i = 0; i < p.size() - 1; i++) {
1952      double delta = cagd::dist(p[i], p[i + 1]);    /* Δi */
1953      sum_delta += delta;
1954      _knot_sqnc.push_back(sum_delta);
1955  }
1956  if (sum_delta < 0.) {    /* Normalize knot sequence so that uL = 1. */
1957      for (knot_itr i = _knot_sqnc.begin(); i < _knot_sqnc.end(); i++) {
1958          *i /= sum_delta;
1959      }
1960  }
```

이 코드는 148번 마디에서 사용된다.

**152.** Centripetal parametrization: 일반적으로 chord length parametrization이 대부분의 경우 잘 동작하지만, 경우에 따라 우리가 원하는 결과가 잘 얻어지지 않는다. 특히 data point가 뾰족한 corner 근방에 놓여 있을 때, chord length parametrization은 그 corner 주변이 둥그스름하게 볼록 솟아나는 곡선을 만들어낸다. 그런 경우 corner의 형상을 올바르게 잡아주려면 centripetal parametrization으로 knot sequence를 생성한다. 이는  $u_{i+1} - u_i = \Delta_i$ 이고  $\|x_{i+1} - x_i\| = \Delta x_i$ 일 때,

$$\frac{\Delta_i}{\Delta_{i+1}} = \left[ \frac{\|\Delta x_i\|}{\|\Delta x_{i+1}\|} \right]^{1/2}$$

가 되도록 knot sequence를 잡아주는 것이며, 결과적으로 곡선을 따라 움직이는 point에 가해지는 구심력 (centripetal force)의 변화(variation)을 부드럽게 만들어준다.

〈Centripetal parametrization of knot sequence 152〉≡

```

1961     double sum_delta = 0.;
1962     _knot_sqnc.push_back(sum_delta);
1963     for (size_t i = 0; i < p.size() - 1; i++) {
1964         double delta = sqrt(cagd::dist(p[i], p[i + 1]));
1965         sum_delta += delta;
1966         _knot_sqnc.push_back(sum_delta);
1967     }
1968     if (sum_delta < 0.) { /* Normalize knot sequence so that u_L = 1. */
1969         for (size_t i = 0; i < _knot_sqnc.size(); i++) {
1970             _knot_sqnc[i] /= sum_delta;
1971         }
1972     }

```

이 코드는 148번 마디에서 사용된다.

**153.** Function spline parametrization: 이는 data point  $x_i$ 의 첫 번째 좌표들을 knot sequence로 설정하는 것이다. 주로 2차원 평면상의 점  $x_i = (u_i, v_i)$ 들이 있을 때,  $u$  축에 대한 함수로서의  $v$ 를 spline interpolation할 때 사용한다.

〈Function spline parametrization of knot sequence 153〉≡

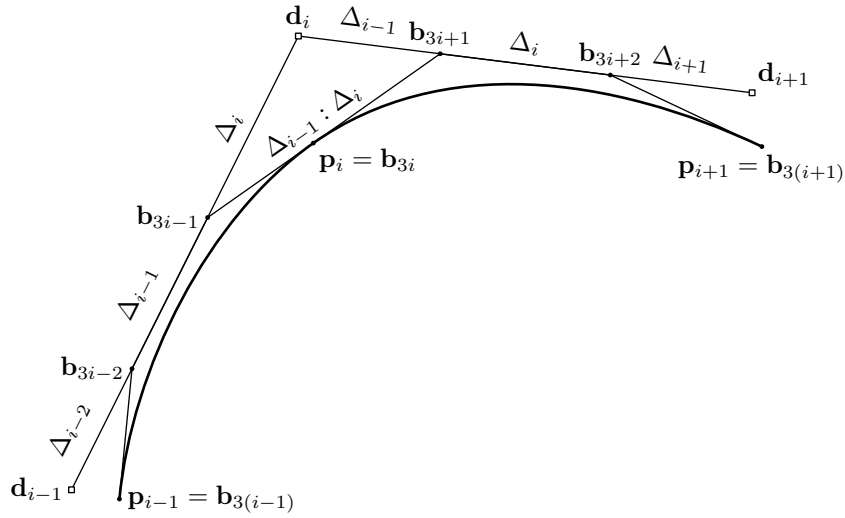
```

1973     for (size_t i = 0; i < p.size(); i++) {
1974         _knot_sqnc.push_back(p[i](1));
1975     }

```

이 코드는 148번 마디에서 사용된다.

**154.** Cubic spline 보간의 해를 구하기 위한 방정식을 유도하기 위하여, 데이터 포인트  $\mathbf{p}_i$ 에서의  $C^2$  연속성 조건을 그림으로 표현하면 아래와 같다.



모든 B-spline은 piecewise Bézier 곡선으로 표현 가능하다. 위의 그림을 참조하면,

$$\mathbf{p}_i = \mathbf{b}_{3i}; \quad i = 0, \dots, L$$

이고, inner Bézier control point,  $\mathbf{b}_{3i\pm 1}$ 과  $\mathbf{p}_i$  사이의 관계는 곡선의  $C^1$  연속성 조건에 의하여

$$\mathbf{p}_i = \frac{\Delta_i \mathbf{b}_{3i-1} + \Delta_{i-1} \mathbf{b}_{3i+1}}{\Delta_{i-1} + \Delta_i}; \quad i = 1, \dots, L-1$$

이다. 이때,  $\Delta_i = \Delta u_i$ 를 간략하게 쓴 것이다. 이제  $C^2$  연속성 조건에 의하여 spline의 컨트롤 포인트  $\mathbf{d}_i$ 와  $\mathbf{b}_{3i\pm 1}$  사이의 관계는

$$\begin{aligned} \mathbf{b}_{3i-1} &= \frac{\Delta_i \mathbf{d}_{i-1} + (\Delta_{i-2} + \Delta_{i-1}) \mathbf{d}_i}{\Delta_{i-2} + \Delta_{i-1} + \Delta_i}; \quad i = 2, \dots, L-1 \\ \mathbf{b}_{3i+1} &= \frac{(\Delta_i + \Delta_{i+1}) \mathbf{d}_i + \Delta_{i-1} \mathbf{d}_{i+1}}{\Delta_{i-1} + \Delta_i + \Delta_{i+1}}; \quad i = 1, \dots, L-2 \end{aligned}$$

이다. 곡선의 양 끝부분에서는 조금 상황이 다르며

$$\begin{aligned} \mathbf{b}_2 &= \frac{\Delta_1 \mathbf{d}_0 + \Delta_0 \mathbf{d}_1}{\Delta_0 + \Delta_1} \\ \mathbf{b}_{3L-2} &= \frac{\Delta_{L-1} \mathbf{d}_{L-1} + \Delta_{L-2} \mathbf{d}_L}{\Delta_{L-2} + \Delta_{L-1}} \\ \mathbf{b}_1 &= \mathbf{d}_0 \\ \mathbf{b}_{3L-1} &= \mathbf{d}_L \end{aligned}$$

이 된다.  $\mathbf{d}_0$ 와  $\mathbf{d}_L$ 은 end condition에 의하여 결정되거나, clamped end condition의 경우에는 임의의 값이 주어진다. 주어진 데이터 포인트  $\mathbf{p}_i$ 와 미지수인 컨트롤 포인트  $\mathbf{d}_i$  사이의 관계식을 정리하면,

$$(\Delta_{i-1} + \Delta_i) \mathbf{p}_i = \alpha_i \mathbf{d}_{i-1} + \beta_i \mathbf{d}_i + \gamma_i \mathbf{d}_{i+1}$$



의 형태가 되며,

$$\begin{aligned}\alpha_i &= \frac{(\Delta_i)^2}{\Delta_{i-2} + \Delta_{i-1} + \Delta_i} \\ \beta_i &= \frac{\Delta_i(\Delta_{i-2} + \Delta_{i-1})}{\Delta_{i-2} + \Delta_{i-1} + \Delta_i} + \frac{\Delta_{i-1}(\Delta_i + \Delta_{i+1})}{\Delta_{i-1} + \Delta_i + \Delta_{i+1}} \\ \gamma_i &= \frac{(\Delta_{i-1})^2}{\Delta_{i-1} + \Delta_i + \Delta_{i+1}}\end{aligned}$$

이다. 이 때,  $\Delta_{-1} = \Delta_L = 0$ 이다.

정리하면 cubic spline 보간의 컨트롤 포인트는 방정식

$$\begin{pmatrix} 1 & & & & & & \\ \alpha_1 & \beta_1 & \gamma_1 & & & & \\ & & & \ddots & & & \\ & & & & \alpha_{L-1} & \beta_{L-1} & \gamma_{L-1} \\ & & & & & & 1 \end{pmatrix} \begin{pmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{L-1} \\ \mathbf{d}_L \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{r}_1 \\ \vdots \\ \mathbf{r}_{L-1} \\ \mathbf{r}_L \end{pmatrix}$$

의 해를 구함으로써 얻을 수 있다. 이 때

$$\begin{aligned}\mathbf{r}_0 &= \mathbf{b}_1 \\ \mathbf{r}_i &= (\Delta_{i-1} + \Delta_i)\mathbf{p}_i \\ \mathbf{r}_L &= \mathbf{b}_{3L-1}\end{aligned}$$

이다.

〈Setup equations of cubic spline interpolation 154〉≡

```

1976  vector<double> a;      /* α, lower diagonal. */
1977  vector<double> b;      /* β, diagonal. */
1978  vector<double> c;      /* γ, upper diagonal. */
1979  vector<point> r;       /* r, right hand side. */
1980  unsigned long L = p.size() - 1;
1981  b.push_back(1.0);      /* First row. */
1982  c.push_back(0.0);
1983  r.push_back(initial);
1984  for (size_t i = 1; i ≠ L; i++) {
1985      double delta_im2 = delta(i - 2);
1986      double delta_im1 = delta(i - 1);
1987      double delta_i = delta(i);
1988      double delta_ip1 = delta(i + 1);
1989      double alpha_i = delta_i * delta_i / (delta_im2 + delta_im1 + delta_i);
1990      double beta_i = delta_i * (delta_im2 + delta_im1) / (delta_im2 + delta_im1 + delta_i) + delta_im1 *
          (delta_i + delta_ip1) / (delta_im1 + delta_i + delta_ip1);
1991      double gamma_i = delta_im1 * delta_im1 / (delta_im1 + delta_i + delta_ip1);
1992      a.push_back(alpha_i);
1993      b.push_back(beta_i);
1994      c.push_back(gamma_i);
1995      r.push_back((delta_im1 + delta_i) * p[i]);

```

```
1996      }
1997      a.push_back(0.);
1998      b.push_back(1.);
1999      r.push_back(end);
```

이 코드는 144번 마디에서 사용된다.

**155.** 앞에서 설명한 바와 같이 cubic spline 보간은 방정식의 갯수보다 미지수의 갯수가 2개 많은 under-constrained system이다. 부족한 조건 2개는 곡선 양 끝단에서 컨트롤 포인트가 만족해야 하는 end condition으로 결정해야하며, **cubic\_spline** 타입은 clamped, Bessel, quadratic, not-a-knot, natural, 그리고 periodic end condition을 지원한다. 아직은 clamped, not-a-knot, 그리고 periodic end condition만을 구현했다.

⟨Modify equations according to end conditions and solve them 155⟩ ≡

```

2000     switch (cond) {
2001     case end_condition::not_a_knot: {
2002         ⟨Modify equations according to not-a-knot end condition 157⟩;
2003     }
2004     case end_condition::clamped: {      /* No modification required. */
2005         vector<point> x(L + 1, point(p[0].dim()));
2006         if (solve_tridiagonal_system(a, b, c, r, x) ≠ 0) {
2007             _err = TRIDIAGONAL_NOT_SOLVABLE;
2008             return;
2009         }
2010         set_control_points(p[0], x, p[L]);
2011     }
2012     break;
2013     case end_condition::periodic: {
2014         ⟨Modify equations according to periodic end condition 158⟩;
2015         vector<point> x(L, point(p[0].dim()));
2016         if (solve_cyclic_tridiagonal_system(a, b, c, r, x) ≠ 0) {
2017             _err = TRIDIAGONAL_NOT_SOLVABLE;
2018             return;
2019         }
2020         point d_plus((((delta(0) + delta(1)) * x[0] + delta(L - 1) * x[1]) / (delta(L - 1) + delta(0) + delta(1))));
2021         point d_minus((((delta(L - 2) + delta(L - 1)) * x[0] + delta(0) * x[L - 1]) / (delta(L - 2) + delta(L -
2022             1) + delta(0))));
2023         vector<point> d(L + 1, point(p[0].dim()));
2024         d[0] = d_plus;
2025         for (size_t i = 1; i ≠ L; i++) {
2026             d[i] = x[i];
2027         }
2028         d[L] = d_minus;
2029         set_control_points(p[0], d, p[L]);
2030     }
2031     break;
2032     default:
2033         _err = UNKNOWN_END_CONDITION;
2034         return;
2035 }

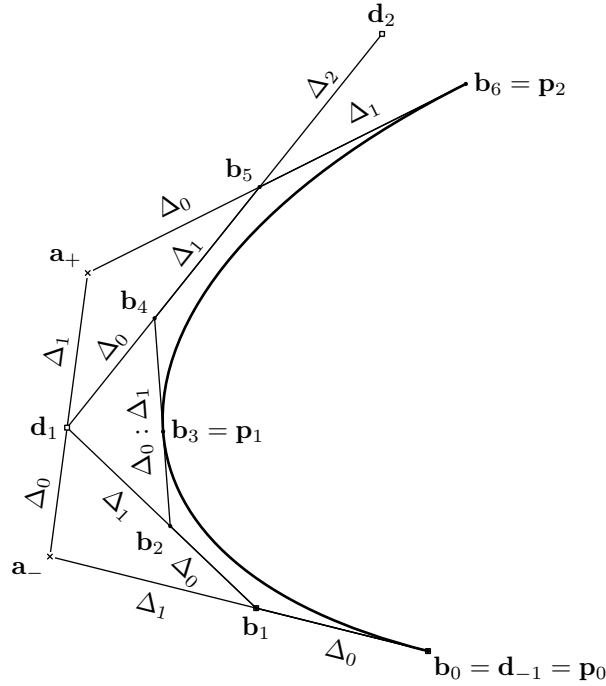
```

이 코드는 144번 마디에서 사용된다.

**156.** 〈Error codes of **cagd** 34〉 +≡

2035 TRIDIAGONAL\_NOT\_SOLVABLE, UNKNOWN\_END\_CONDITION ,

**157.** Not-a-knot end condition은 곡선 양 끝에 놓인 각각 2개의 곡선 조각들이 하나의 Bézier 곡선이 되도록 하는 조건이다. 보간해야 하는 데이터 포인트,  $\mathbf{p}_0, \dots, \mathbf{p}_L$  이 있으면,  $\mathbf{p}_0$ 과  $\mathbf{p}_1$ 을 연결하는 곡선과  $\mathbf{p}_1$ 과  $\mathbf{p}_2$ 를 연결하는 곡선이  $\mathbf{p}_0$ 과  $\mathbf{p}_2$ 를 연결하는 한 곡선의 subdivision이 되도록 하는 것이다. 이는  $\mathbf{p}_{L-2}, \mathbf{p}_{L-1}, \mathbf{p}_L$  사이에서도 동일하게 주어지는 조건이다. 아래의 그림은 not-a-knot end condition을 만족하는 곡선의 시작 부분을 보여준다.



먼저  $\mathbf{p}_0$ 부터  $\mathbf{p}_2$ 까지 하나의 Bézier 곡선이 되어야 하는 조건은 de Casteljau 알고리즘으로부터

$$\mathbf{d}_0 = (1-s)\mathbf{p}_0 + s\mathbf{a}_-; \quad s = \frac{\Delta_0}{\Delta_0 + \Delta_1}$$

$$\mathbf{b}_5 = (1-s)\mathbf{a}_+ + s\mathbf{p}_2 = (1-r)\mathbf{d}_1 + r\mathbf{d}_2; \quad r = \frac{\Delta_0 + \Delta_1}{\Delta_0 + \Delta_1 + \Delta_2}$$

$$\mathbf{d}_1 = (1-s)\mathbf{a}_- + s\mathbf{a}_+$$

이므로

$$\begin{aligned} \mathbf{a}_- &= \frac{1}{s}\mathbf{d}_0 - \frac{1-s}{s}\mathbf{p}_0 \\ \mathbf{a}_+ &= \frac{1}{1-s} \left\{ (1-r)\mathbf{d}_1 + r\mathbf{d}_2 \right\} - \frac{s}{1-s}\mathbf{p}_2 \end{aligned}$$

을 세 번째 식에 대입하고 정리하면

$$\frac{1-s}{s}\mathbf{d}_0 + \left( \frac{2s-sr-1}{1-s} \right) \mathbf{d}_1 + \frac{sr}{1-s}\mathbf{d}_2 = \frac{(1-s)^2}{s}\mathbf{p}_0 + \frac{s^2}{1-s}\mathbf{p}_2 \quad (*)$$

다.

한편,  $\mathbf{p}_1$ 에서의  $C^2$  연속성 조건을 기술하면,

$$\mathbf{b}_2 = s\mathbf{d}_1 + (1-s)\mathbf{d}_0;$$

$$\mathbf{b}_4 = q\mathbf{d}_1 + (1-q)\mathbf{d}_2; \quad q = \frac{\Delta_1 + \Delta_2}{\Delta_0 + \Delta_1 + \Delta_2}$$

$$\mathbf{p}_1 = (1-s)\mathbf{b}_2 + s\mathbf{b}_4$$

이므로

$$(1-s)^2 \mathbf{d}_0 + s(1-s+q) \mathbf{d}_1 + s(1-q) \mathbf{d}_2 = \mathbf{p}_1$$

이다. 이때,  $q = 1 - sr$ 이므로  $q$ 를 소거하면

$$(1-s)^2 \mathbf{d}_0 + s(2-s-sr) \mathbf{d}_1 + s^2 r \mathbf{d}_2 = \mathbf{p}_1 \quad (**)$$

이 된다.  $\mathbf{d}_2$  항을 소거하여 tridiagonal matrix 방정식을 얻기 위해 식 (\*\*)를  $(**) - (*) \times s(1-s)$ 로 치환하면,

$$(-3s^2 + 3s) \mathbf{d}_1 = -(1-s)^3 \mathbf{p}_0 + \mathbf{p}_1 - s^3 \mathbf{p}_2$$

이다. 곡선의 마지막 부분 두 개의 조각에 대해서도 같은 과정을 통하여 방정식을 유도할 수 있다.

정리하면,

$$\begin{pmatrix} 0 & -3s_i^2 + 3s_i & & & \\ \frac{1-s_i}{s_i} & \frac{s_i}{1-s_i}(1-r_i) - 1 & \frac{s_i r_i}{1-s_i} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{s_f r_f}{1-s_f} & \frac{s_f}{1-s_f}(1-r_f) - 1 & \frac{1-s_f}{s_f} \\ & & & -3s_f^2 + 3s_f & 0 \end{pmatrix} \begin{pmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{L-1} \\ \mathbf{d}_L \end{pmatrix} = \begin{pmatrix} -(1-s_i)^3 \mathbf{p}_0 + \mathbf{p}_1 - s_i^3 \mathbf{p}_2 \\ \frac{(1-s_i)^2}{s_i} \mathbf{p}_0 + \frac{s_i^2}{1-s_i} \mathbf{p}_2 \\ \vdots \\ \frac{s_f^2}{1-s_f} \mathbf{p}_{L-2} + \frac{(1-s_f)^2}{s_f} \mathbf{p}_L \\ -s_f^3 \mathbf{p}_{L-2} + \mathbf{p}_{L-1} - (1-s_f)^3 \mathbf{p}_L \end{pmatrix}$$

이다. 위의 식에서

$$\begin{aligned} s_i &= \frac{\Delta_0}{\Delta_0 + \Delta_1} \\ r_i &= \frac{\Delta_0 + \Delta_1}{\Delta_0 + \Delta_1 + \Delta_2} \\ s_f &= \frac{\Delta_{L-1}}{\Delta_{L-2} + \Delta_{L-1}} \\ r_f &= \frac{\Delta_{L-2} + \Delta_{L-1}}{\Delta_{L-3} + \Delta_{L-2} + \Delta_{L-1}} \end{aligned}$$

이다.

중간 부분은 앞의 cubic spline 보간에 관한 방정식의  $\alpha_i$ ,  $\beta_i$ ,  $\gamma_i$ 와  $r_i$ 를 그대로 채워 넣는다. Not-a-knot end condition은 최소 3개 이상( $2 \leq L$ )이어야 적용 가능하다. 특히  $L = 2$ 인 경우에는  $s_i = \Delta_0/(\Delta_0 + \Delta_1)$ ,  $s_f = \Delta_1/(\Delta_0 + \Delta_1)$ ,  $r_i = r_f = 1$ 이 되어 경계 조건의 방정식은

$$\begin{pmatrix} 0 & \frac{3\Delta_0\Delta_1}{(\Delta_0+\Delta_1)^2} & \\ \frac{\Delta_1}{\Delta_0} & -1 & \frac{\Delta_0}{\Delta_1} \\ & \frac{3\Delta_0\Delta_1}{(\Delta_0+\Delta_1)^2} & 0 \end{pmatrix} \begin{pmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \mathbf{d}_2 \end{pmatrix} = \begin{pmatrix} -\frac{\Delta_1^3}{(\Delta_0+\Delta_1)^3} \mathbf{p}_0 + \mathbf{p}_1 - \frac{\Delta_0^3}{(\Delta_0+\Delta_1)^3} \mathbf{p}_2 \\ \frac{\Delta_1^2}{\Delta_0(\Delta_0+\Delta_1)} \mathbf{p}_0 + \frac{\Delta_0^2}{\Delta_1(\Delta_0+\Delta_1)} \mathbf{p}_2 \\ -\frac{\Delta_1^3}{(\Delta_0+\Delta_1)^3} \mathbf{p}_0 + \mathbf{p}_1 - \frac{\Delta_0^3}{(\Delta_0+\Delta_1)^3} \mathbf{p}_2 \end{pmatrix}$$

이 된다.

$\langle \text{Modify equations according to not-a-knot end condition } 157 \rangle \equiv$

2036 if ( $L \geq 2$ ) {

```

2037     double s_i = delta(0)/(delta(0) + delta(1));
2038     double r_i = (delta(0) + delta(1))/(delta(0) + delta(1) + delta(2));
2039     double s_f = delta(L - 1)/(delta(L - 2) + delta(L - 1));
2040     double r_f = (delta(L - 2) + delta(L - 1))/(delta(L - 3) + delta(L - 2) + delta(L - 1));
2041     b[0] = 0.;    /* First row. */
2042     c[0] = -3 * s_i * s_i + 3 * s_i;
2043     r[0] = -(1 - s_i) * (1 - s_i) * (1 - s_i) * p[0] + p[1] - s_i * s_i * s_i * p[2];
2044     a[0] = (1 - s_i)/s_i;    /* Second row. */
2045     b[1] = s_i/(1 - s_i) * (1 - r_i) - 1;
2046     c[1] = s_i * r_i/(1 - s_i);
2047     r[1] = (1 - s_i) * (1 - s_i)/s_i * p[0] + s_i * s_i/(1 - s_i) * p[2];
2048     a[L - 2] = s_f * r_f/(1 - s_f);    /* Second to the last row. */
2049     b[L - 1] = s_f/(1 - s_f) * (1 - r_f) - 1;
2050     c[L - 1] = (1 - s_f)/s_f;
2051     r[L - 1] = s_f * s_f/(1 - s_f) * p[L - 2] + (1 - s_f) * (1 - s_f)/s_f * p[L];
2052     a[L - 1] = -3 * s_f * s_f + 3 * s_f;    /* Last row. */
2053     b[L] = 0.;
2054     r[L] = -s_f * s_f * s_f * p[L - 2] + p[L - 1] - (1 - s_f) * (1 - s_f) * (1 - s_f) * p[L];
2055     }

```

이 코드는 155번 마디에서 사용된다.

**158.** 사람의 보행궤적과 같은 주기적인 운동궤적을 다루기 위해서는 곡선의 시작점과 끝점이 일치( $\mathbf{p}_0 = \mathbf{p}_L$ )할 뿐 아니라 그 점에서 2차 미분까지 연속( $C^2$  condition)인 곡선이 필요하다. 이 때의 컨트롤 포인트는 방정식

$$\begin{pmatrix} \beta_0 & \gamma_0 & & & \alpha_0 \\ \alpha_1 & \beta_1 & \gamma_1 & & \\ & & \ddots & & \\ & & & \alpha_{L-2} & \beta_{L-2} & \gamma_{L-2} \\ \gamma_{L-1} & & & \alpha_{L-1} & \beta_{L-1} \end{pmatrix} \begin{pmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{L-1} \end{pmatrix} = \begin{pmatrix} \mathbf{r}_0 \\ \mathbf{r}_1 \\ \vdots \\ \mathbf{r}_{L-1} \end{pmatrix}$$

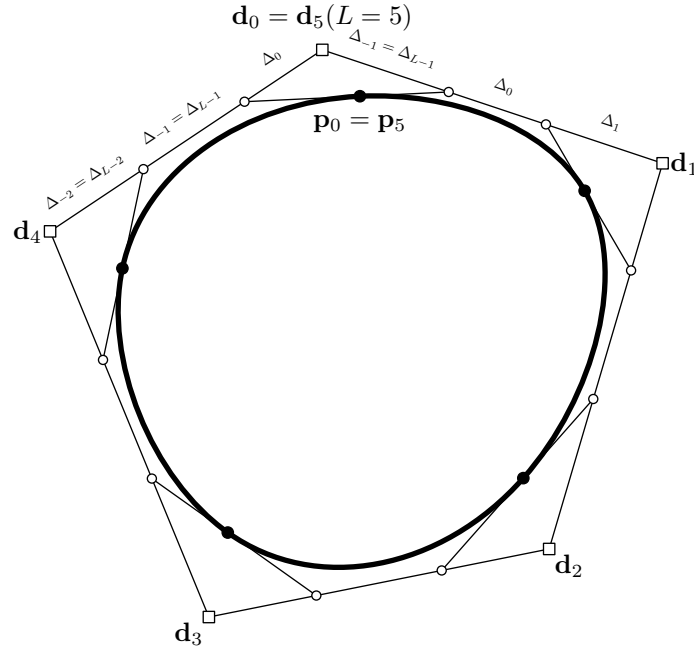
의 해를 구함으로써 얻을 수 있다. 이 때

$$\mathbf{r}_i = (\Delta_{i-1} + \Delta_i)\mathbf{p}_i$$

이고,

$$\Delta_0 = \Delta_L, \quad \Delta_{-1} = \Delta_{L-1}, \quad \Delta_{-2} = \Delta_{L-2}$$

이다.  $\alpha_i, \beta_i, \gamma_i, \mathbf{r}_i$ 의 정의는 앞의  $C^2$  조건으로부터 유도되는 식을 따르는데, 새로운  $\Delta_i$ 의 정의로 인하여  $\alpha_0, \alpha_1, \beta_0, \beta_1, \beta_{L-1}, \gamma_0, \gamma_{L-1}, \mathbf{r}_0$ 는 새로 계산해야한다. 아래 그림은  $L = 5$ 인 경우의 periodic end condition을 보여준다.



〈 Modify equations according to periodic end condition 158 〉 ≡

```

2056   for (size_t i = L - 1; i != 0; i--) { /* Modify  $\alpha$ . */
2057       a[i] = a[i - 1];
2058   }
2059   a[0] = delta(0) * delta(0) / (delta(L - 2) + delta(L - 1) + delta(0));
2060   a[1] = delta(1) * delta(1) / (delta(L - 1) + delta(0) + delta(1));
2061   b.pop_back(); /* Modify  $\beta$ . */
2062   b[0] = delta(0) * (delta(L - 2) + delta(L - 1)) / (delta(L - 2) + delta(L - 1) + delta(0)) + delta(L - 1) *
        (delta(0) + delta(1)) / (delta(L - 1) + delta(0) + delta(1));

```



```

2063       $b[1] = \text{delta}(1) * (\text{delta}(L-1) + \text{delta}(0)) / (\text{delta}(L-1) + \text{delta}(0) + \text{delta}(1)) + \text{delta}(0) * (\text{delta}(1) +$ 
       $\text{delta}(2)) / (\text{delta}(0) + \text{delta}(1) + \text{delta}(2));$ 
2064       $b[L-1] = \text{delta}(L-1) * (\text{delta}(L-3) + \text{delta}(L-2)) / (\text{delta}(L-3) + \text{delta}(L-2) + \text{delta}(L-1)) +$ 
       $\text{delta}(L-2) * (\text{delta}(L-1) + \text{delta}(0)) / (\text{delta}(L-2) + \text{delta}(L-1) + \text{delta}(0));$ 
2065       $c[0] = \text{delta}(L-1) * \text{delta}(L-1) / (\text{delta}(L-1) + \text{delta}(0) + \text{delta}(1));$       /* Modify  $\gamma$ . */
2066       $c[L-1] = \text{delta}(L-2) * \text{delta}(L-2) / (\text{delta}(L-2) + \text{delta}(L-1) + \text{delta}(0));$ 
2067       $r.\text{pop\_back}();$       /* Modify  $\mathbf{r}$ . */
2068       $r[0] = (\text{delta}(L-1) + \text{delta}(0)) * p[0];$ 

```

이 코드는 155번 마디에서 사용된다.

**159. Test: Cubic Spline Interpolation.**

$x = \pi, \pi + 1, \dots, \pi + 10$ 일 때,  $y = \sin(x) + 3$ 으로 주어지는 data point,

$$y = 3.0000, 2.1585, 2.0907, 2.8589, 3.7568, 3.9589, 3.2794, 2.3430, 2.0106, 2.5879, 3.5440$$

을 cubic spline으로 보간하는 예제를 보여준다. End condition은 not-a-knot을 적용한다.

⟨ Test routines 26 ⟩ +=

```

2069   print_title("cubic_spline_interpolation");
2070   {
2071       ⟨ Generate example data points 160 ⟩;
2072       cubic_spline crv(p, cubic_spline::end_condition::not_a_knot,
                        cubic_spline::parametrization::function_spline);
2073       psf file = create_postscript_file("sine_curve.ps");
2074       crv.write_curve_in_postscript(file, 100, 1., 1, 2, 40.);
2075       crv.write_control_polygon_in_postscript(file, 1., 1, 2, 40.);
2076       crv.write_control_points_in_postscript(file, 1., 1, 2, 40.);
2077       close_postscript_file(file, true);
2078       ⟨ Compare the result of interpolation with MATLAB 161 ⟩;
2079       cout << crv.description();
2080       const unsigned steps = 1000;
2081       vector<double> knots = crv.knot_sequence();
2082       double du = (knots[knots.size() - 3] - knots[2]) / double(steps - 1);
2083       double us[steps];
2084       vector<point> crv_pts_s(steps, point(2));
2085       for (size_t i = 0; i < steps; i++) {
2086           us[i] = knots[2] + i * du;
2087       }
2088       auto t0 = high_resolution_clock::now();
2089       for (size_t i = 0; i < steps; i++) {
2090           crv_pts_s[i] = crv.evaluate(us[i]);
2091       }
2092       auto t1 = high_resolution_clock::now();
2093       cout << "Serial_computation: " << duration_cast<milliseconds>(t1 - t0).count() << " msec\n";
2094       t0 = high_resolution_clock::now();
2095       vector<point> crv_pts_p = crv.evaluate_all(steps);
2096       t1 = high_resolution_clock::now();
2097       cout << "Parallel_computation: " << duration_cast<milliseconds>(t1 - t0).count() <<
           " msec\n";
2098       double diff = 0.;
2099       for (size_t i = 0; i < steps; i++) {

```

```

2100     diff += dist(crv_pts_s[i], crv_pts_p[i]);
2101 }
2102 cout << "Mean_difference_between_serial_and_parallel_computation=" <<
      diff/double(steps) << endl;
2103 }

```

**160.** 〈Generate example data points 160〉 ≡

```

2104 vector<point> p;
2105 for (unsigned i = 0; i < 11; i++) {
2106     point datum = point({0, 0});
2107     datum(1) = static_cast<double>(i) + M_PI;
2108     datum(2) = sin(datum(1)) + 3.;
2109     p.push_back(datum);
2110 }

```

이 코드는 159번 마디에서 사용된다.

**161.** 보간 결과의 정확성을 검증하기 위하여 MATLAB에서 *spline()* 함수를 이용하여 보간한 것과 비교한다.  $x = \pi, \pi + .25, \pi + .5, \dots, \pi + 10$ 에 대하여 cubic spline으로 보간한 함수  $f()$ 로부터  $y = f(x)$ 를 계산한 것을 root-mean-square로 상호 비교했다.

〈Compare the result of interpolation with MATLAB 161〉 ≡

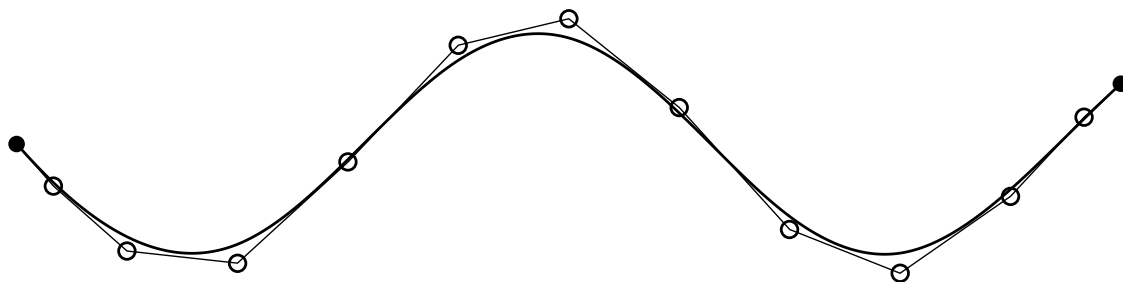
```

2111 double matlab_bench[] = {3.0000, 2.7308, 2.4983, 2.3062, 2.1585, 2.0592, 2.0122, 2.0214, 2.0907, 2.2211,
      2.4018, 2.6190, 2.8589, 3.1075, 3.3501, 3.5715, 3.7568, 3.8928, 3.9742, 3.9974, 3.9589, 3.8578, 3.7032,
      3.5065, 3.2794, 3.0342, 2.7858, 2.5502, 2.3430, 2.1785, 2.0643, 2.0063, 2.0106, 2.0804, 2.2073, 2.3802,
      2.5879, 2.8193, 3.0632, 3.3085, 3.5440};
2112 double interpolated[41];
2113 double u = M_PI;
2114 double err = 0.;
2115 for (size_t i = 0; i < 41; i++) {
2116     double y = crv.evaluate(u)(2);
2117     interpolated[i] = y;
2118     u += 0.25;
2119     err += (interpolated[i] - matlab_bench[i]) * (interpolated[i] - matlab_bench[i]);
2120 }
2121 err /= 41;
2122 err = sqrt(err);
2123 cout << "RMS_error_of_interpolation_(compared_with_MATLAB)=" << err << endl;

```

이 코드는 159번 마디에서 사용된다.

## 162. 실행 결과.

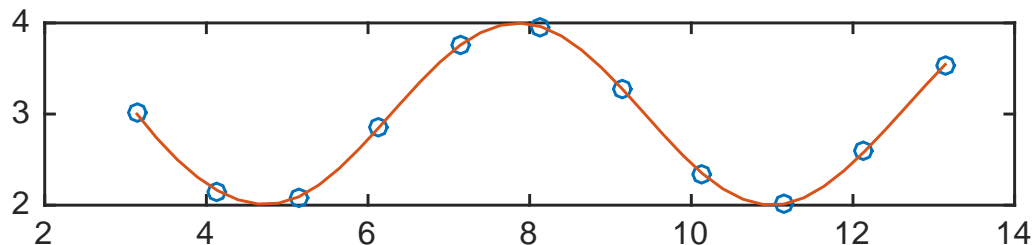


MATLAB에서 *spline()* 함수를 이용하여 같은 데이터를 cubic spline 보간한 것과 결과를 비교하면 다음과 같은 결과가 출력된다. 오차가  $10^{-5}$  오더로 발생한 것은 MATLAB에서 계산한 결과를 소수점 4째 자리까지 반올림한 것으로 가져왔기 때문이다.

RMS error of interpolation (compared with MATLAB) = 2.29482e-05

## 163. 참고로 MATLAB에서 같은 데이터로 cubic spline 보간을 하는 코드와 결과는 다음과 같다.

```
>> x = pi:1:(pi+10);
>> y = sin(x)+3;
>> xx = pi:.25:pi+10;
>> yy = spline (x, y, xx);
>> plot (x, y, 'o', xx, yy);
>> set (gcf, 'PaperPosition', [0 0 10 2]);
>> set (gcf, 'PaperSize', [10 2]);
>> saveas (gcf, 'matlab', 'pdf')
```



**164. Test: Cubic Spline Interpolation (Degenerate Case).**

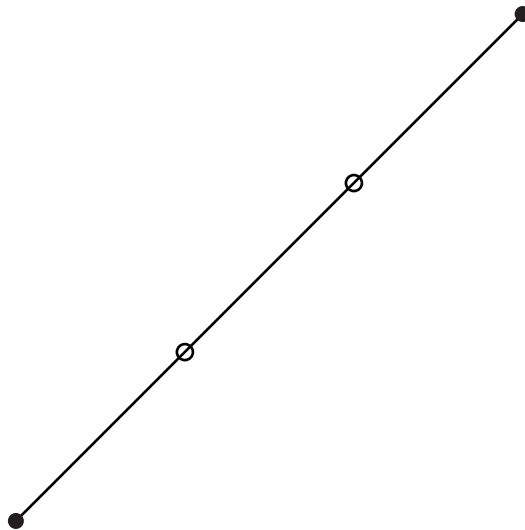
단 두개의 데이터 포인트에 대한 cubic spline interpolation을 테스트한다. (10, 10)과 (200, 200)을 연결하는 cubic spline interpolation을 not-a-knot end condition으로 구한다. 두 개의 데이터 포인트로는 not-a-knot end condition이 성립될 수 없는 degenerate case이며 두 점을 연결하는 직선이 나와야 한다.

〈Test routines 26〉 +=

```

2124   print_title("cubic_spline_interpolation:_degenerate_case");
2125   {
2126       vector<point> p;
2127       p.push_back(point({10, 10}));
2128       p.push_back(point({200, 200}));
2129       cubic_spline crv(p);
2130       psf file = create_postscript_file("line.ps");
2131       crv.write_curve_in_postscript(file, 30, 1., 1, 2, 1.);
2132       crv.write_control_polygon_in_postscript(file, 1., 1, 2, 1.);
2133       crv.write_control_points_in_postscript(file, 1., 1, 2, 1.);
2134       close_postscript_file(file, true);
2135   }

```

**165. 실행 결과.**

**166. Test: Periodic Cubic Spline Interpolation.**

$$p_i = r(\cos 2\pi i/6, \sin 2\pi i/6), \quad (i = 0, \dots, 6)$$

을 periodic cubic spline으로 보간한다.

〈Test routines 26〉 +≡

```

2136   print_title("periodic_spline_interpolation"); { vector<point> p;
2137   double r = 100.;
2138   cout << "Data_points:" << endl;
2139   for (size_t i = 0; i < 7; i++) {
2140       p.push_back(point({r * cos(2 * M_PI/6 * i) + 200., r * sin(2 * M_PI/6 * i) + 200.}));
2141       cout << " (" << r * cos(2 * M_PI/6 * i) + 200. << ", " << r * sin(2 * M_PI/6 * i) + 200. << ") " << endl;
2142   }
2143   cubic_spline crv(p, cubic_spline::end_condition::periodic,
                     cubic_spline::parametrization::centripetal);
2144   psf file = create_postscript_file("periodic.ps");
2145   crv.write_curve_in_postscript(file, 200, 1., 1, 2, 1.);
2146   crv.write_control_polygon_in_postscript(file, 1., 1, 2, 1.);
2147   crv.write_control_points_in_postscript(file, 1., 1, 2, 1.);
2148   close_postscript_file(file, true);
2149   cout << crv.description();
2150   const unsigned steps = 1000;
2151   vector<double> knots = crv.knot_sequence();
2152   double du = (knots[knots.size() - 3] - knots[2]) / double(steps - 1);
2153   double us[steps];
2154   vector<point> crv_pts_s(steps, point(2));
2155   for (size_t i = 0; i < steps; i++) {
2156       us[i] = knots[2] + i * du;
2157   }
2158   auto t0 = high_resolution_clock::now();
2159   for (size_t i = 0; i < steps; i++) {
2160       crv_pts_s[i] = crv.evaluate(us[i]);
2161   }
2162   auto t1 = high_resolution_clock::now();
2163   cout << "Serial_computation: " << duration_cast<milliseconds>(t1 - t0).count() << " msec\n";
2164   t0 = high_resolution_clock::now();
2165   vector<point> crv_pts_p = crv.evaluate_all(steps);
2166   t1 = high_resolution_clock::now();
2167   cout << "Parallel_computation: " << duration_cast<milliseconds>(t1 - t0).count() << " msec\n";
       double error = 0.; for (size_t i = 0; i < steps; i++) { error += dist(crv_pts_s[i], crv_pts_p[i]);

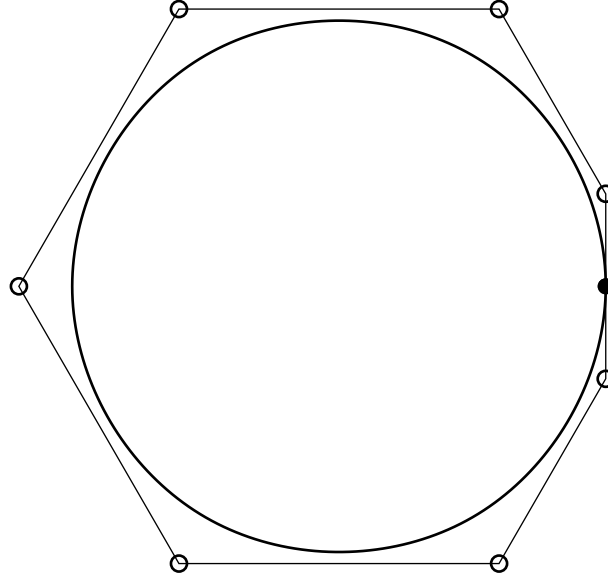
```

```

} cout << "Mean_difference_between_serial_and_parallel_computation=" << error /
double(steps) << endl; }

```

167. 실행 결과.



168.  $C^2$  cubic spline 곡선은 knot에서 나누는 각 조각별로 형상이 같은 Bézier 곡선으로 변환할 수 있다.

〈Methods to obtain a **bezier** curve for a segment of **cubic\_spline** 168〉  $\equiv$

```

2168 void cubic_spline::bezier_control_points(vector<point> &bezier_ctrl_points, vector<double> &knot)
      const {
2169     bezier_ctrl_points.clear();
2170     knot.clear();
2171     〈Create a new knot sequence of which each knot has multiplicity of 1 170〉;
2172     〈Check whether the curve can be broken into Bézier curves 171〉;
2173     〈Calculate Bézier control points 172〉;
2174 }

```

이 코드는 101번 마디에서 사용된다.

169. 〈Error codes of **cagd** 34〉  $\equiv$

```

2175 UNABLE_TO_BREAK_INTO_BEZIER ,

```

**170.** 모든 knot들의 multiplicity가 1이 되도록 한다. Knot sequence를 따라가며 순증가하는 knot들만 추려낸다.

〈Create a new knot sequence of which each knot has multiplicity of 1 170〉≡

```

2176     knot.push_back(_knot_sqnc[0]);
2177     for (size_t i = 1; i < _knot_sqnc.size(); i++) {
2178         if (_knot_sqnc[i] > knot.back()) {
2179             knot.push_back(_knot_sqnc[i]);
2180         }
2181     }

```

이 코드는 168번 마디에서 사용된다.

**171.** 모든 knot들의 multiplicity가 1이 되도록 만든 후 knot의 갯수와 control point들의 갯수를 비교함으로써 cubic spline curve를 Bézier curve로 변환 가능한지 점검한다.

〈Check whether the curve can be broken into Bézier curves 171〉≡

```

2182     if (knot.size() + 2 < _ctrl_pts.size()) {
2183         _err = UNABLE_TO_BREAK_INTO_BEZIER;
2184         return;
2185     }

```

이 코드는 168번 마디에서 사용된다.



**172.** 먼저 필요한 저장공간을 확보한 후, 각 곡선의 segment별로 Bézier 컨트롤 포인트를 계산한다.

〈Calculate Bézier control points 172〉≡

```

2186   for (size_t i = 0; i ≤ 3 * (knot.size() - 1); i++) {
2187       bezier_ctrl_points.push_back(point({0.0, 0.0}));
2188   }
2189   bezier_ctrl_points[0] = _ctrl_pts[0];    /* Special treatment on the first segment. */
2190   bezier_ctrl_points[1] = _ctrl_pts[1];
2191   double delta = knot[2] - knot[0];
2192   bezier_ctrl_points[2] = ((knot[2] - knot[1]) * _ctrl_pts[1] + (knot[1] - knot[0]) * _ctrl_pts[2]) / delta;
2193   for (size_t i = 2; i ≤ knot.size() - 2; i++) {    /* Intermediate segments. */
2194       delta = knot[i + 1] - knot[i - 2];
2195       bezier_ctrl_points[3 * i - 1] = ((knot[i + 1] - knot[i]) * _ctrl_pts[i] + (knot[i] - knot[i - 2]) * _ctrl_pts[i + 1]) / delta;
2196       bezier_ctrl_points[3 * i - 2] = ((knot[i + 1] - knot[i - 1]) * _ctrl_pts[i] + (knot[i - 1] - knot[i - 2]) *
        _ctrl_pts[i + 1]) / delta;
2197   }
2198   unsigned long L = knot.size() - 1;    /* Special treatment on the last segment. */
2199   delta = knot[L] - knot[L - 2];
2200   bezier_ctrl_points[3 * L - 2] = ((knot[L] - knot[L - 1]) * _ctrl_pts[L] + (knot[L - 1] - knot[L - 2]) *
        _ctrl_pts[L + 1]) / delta;
2201   bezier_ctrl_points[3 * L - 1] = _ctrl_pts[L + 1];
2202   bezier_ctrl_points[3 * L] = _ctrl_pts[L + 2];
2203   for (size_t i = 1; i ≤ (knot.size() - 2); i++) {    /* Finally, calculate  $b_{3i}$ s. */
2204       delta = knot[i + 1] - knot[i - 1];
2205       bezier_ctrl_points[3 * i] = ((knot[i + 1] - knot[i]) * bezier_ctrl_points[3 * i - 1] + (knot[i] - knot[i - 1]) *
        bezier_ctrl_points[3 * i + 1]) / delta;
2206   }

```

이 코드는 168번 마디에서 사용된다.

**173.** 〈Methods of cubic\_spline 103〉 +≡

```

2207   protected:
2208       void bezier_control_points(vector<point> &, vector<double> &) const;

```

**174.** Cubic spline 곡선의 곡률은 먼저 곡선을 Bézier 곡선으로 변환한 후, Bézier 곡선의 곡률을 계산함으로써 구한다.

〈Methods to calculate curvature of **cubic\_spline** 174〉 ≡

```

2209     vector<point> cubic_spline::signed_curvature(int density) const {
2210         vector<point> bezier_ctrl_points;
2211         vector<double> knot;
2212         vector<point> curvature;
2213         bezier_control_points(bezier_ctrl_points, knot);    /* Get equivalent Bézier curves. */
2214         for (size_t i = 0; i < knot.size() - 2; i++) {
2215             list<point> cpts;    /* Control points for a section of Bézier curve. */
2216             cpts.clear();
2217             cpts.push_back(bezier_ctrl_points[3 * i]);
2218             cpts.push_back(bezier_ctrl_points[3 * i + 1]);
2219             cpts.push_back(bezier_ctrl_points[3 * i + 2]);
2220             cpts.push_back(bezier_ctrl_points[3 * i + 3]);
2221             bezier segment(cpts);
2222             vector<point> kappa = segment.signed_curvature(density);
2223             for (size_t j = 0; j < kappa.size(); j++) {
2224                 curvature.push_back(kappa[j]);
2225             }
2226         }
2227         return curvature;
2228     }

```

이 코드는 101번 마디에서 사용된다.

**175.** 〈Methods of **cubic\_spline** 103〉 +≡

```

2229     protected:
2230         vector<point> signed_curvature(int) const;

```

**176.** 먼저 knot insertion을 수행하는 method를 정의한다. Knot insertion은 새로 삽입된 knot에 의하여 Greville abscissas를 새로 계산하고, 그에 따라 컨트롤 포인트들을 linear interpolation하는 과정이다.

먼저 삽입할 knot이 적절한 범위의 값인지 점검한다. 그리고 새로 삽입하는 knot의 영향을 받지 않는 컨트롤 포인트들을 새로운 저장공간에 복사한다. 새로 계산해야하는 컨트롤 포인트를 linear interpolation으로 계산하고, 다시 새로운 knot의 영향을 받지 않는 나머지 컨트롤 포인트들을 복사한다.

마지막으로 주어진 knot을 *\_knot\_sqnc*에 삽입하고, 새로 계산한 컨트롤 포인트들로 *\_ctrl\_pts*를 대체한다.

⟨Methods for knot insertion and removal of **cubic\_spline** 176⟩ ≡

```

2231 void cubic_spline::insert_knot(const double u) {
2232     const int n = 3;      /* Degree of cubic spline. */
2233     size_t index = find_index_in_knot_sequence(u);
2234     if (index ≡ SIZE_MAX) {
2235         _err = OUT_OF_KNOT_RANGE;
2236     }
2237     if ((index < n - 1) ∨ (int(_knot_sqnc.size()) - n < index)) {
2238         _err = NOT_INSERTABLE_KNOT;
2239     }
2240     vector<point> new_ctrl_pts;    /* construct a new control points */
2241     ⟨Copy control points for  $i = 0, \dots, I - d + 1$  178⟩;
2242     ⟨Construct new control points by piecewise linear interpolation 179⟩;
2243     ⟨Copy remaining control points to new control points 180⟩;
2244     _knot_sqnc.insert(_knot_sqnc.begin() + index + 1, u);
2245     _ctrl_pts.clear();
2246     _ctrl_pts = new_ctrl_pts;
2247 }
```

184번 마디도 살펴보자.

이 코드는 101번 마디에서 사용된다.

**177.** ⟨Error codes of **cagd** 34⟩ +≡

```

2248 NOT_INSERTABLE_KNOT ,
```

**178.** ⟨Copy control points for  $i = 0, \dots, I - d + 1$  178⟩ ≡

```

2249 for (size_t i = 0; i ≤ index - n + 1; i++) {
2250     new_ctrl_pts.push_back(_ctrl_pts[i]);
2251 }
```

이 코드는 176번 마디에서 사용된다.

**179.** ⟨Construct new control points by piecewise linear interpolation 179⟩ ≡

```

2252 for (size_t i = index - n + 2; i ≤ index + 1; i++) {
2253     new_ctrl_pts.push_back(_ctrl_pts[i - 1] * (_knot_sqnc[i + n - 1] - u) / (_knot_sqnc[i + n - 1] - _knot_sqnc[i - 1])
2254                             + _ctrl_pts[i] * (u - _knot_sqnc[i - 1]) / (_knot_sqnc[i + n - 1] - _knot_sqnc[i - 1]));
2254 }
```

이 코드는 176번 마디에서 사용된다.

**180.**  $\langle$  Copy remaining control points to new control points 180  $\rangle \equiv$

```
2255     for (size_t i = index + 2; i ≤ _knot_sqnc.size() - n + 1; i++) {
2256         new_ctrl_pts.push_back(_ctrl_pts[i - 1]);
2257     }
```

이 코드는 176번 마디에서 사용된다.

**181.**  $\langle$  Methods of **cubic\_spline** 103  $\rangle + \equiv$

```
2258     public:
2259         void insert_knot(const double);
```

**182.** Knot removal을 구현하기 위하여 몇 가지 method를 먼저 정의한다. *get\_blending\_ratio()*는 Eck의 알고리즘에서 언급하는 blending ratio를 계산한다. *bracket()*과 *find\_l()*은 Eck의 논문에서 사용하는 notation을 구현한 것이다.

⟨Miscellaneous methods of **cubic\_spline** 116⟩ +=

```

2260  double cubic_spline::get_blending_ratio(const vector<double> &IGESKnot, long v, long r, long i)
      {
2261      long beta = 1;      /* set beta and determine m1 and m2 */
2262      long m1 = beta - r + 6 - v;
2263      if (m1 < 0) {
2264          m1 = 0;
2265      }
2266      long m2 = r - _ctrl_pts.size() + 2 + beta;
2267      if (m2 < 0) {
2268          m2 = 0;
2269      }
2270      if ((v - 1 ≤ i) ∧ (i ≤ v - 2 + m1)) {      /* special cases to return 0 or 1 */
2271          return 0.;
2272      }
2273      if ((4 - m2 ≤ i) ∧ (i ≤ 3)) {
2274          return 1.;
2275      }
2276      double gamma = 0.;      /* otherwise go through a laborious chore */
2277      for (size_t j = v - 1 + m1; j ≤ 4 - m2; j++) {
2278          double brk = bracket(IGESKnot, j + 1, 3, r);
2279          gamma += brk * brk;
2280      }
2281      double result = 0.;
2282      for (size_t j = v - 1 + m1; j ≤ i; j++) {
2283          double brk = bracket(IGESKnot, j + 1, 3, r);
2284          result += brk * brk;
2285      }
2286      return result / gamma;
2287  }

2288  double cubic_spline::bracket(const vector<double> &IGESKnot, long a, long b, long r) {
2289      if (a ≡ b + 1) {
2290          return 1./find_l(IGESKnot, a - 1, r);
2291      }
2292      if (a ≡ b + 2) {
2293          return 1./(1. - find_l(IGESKnot, a - 1, r));
2294      }

```

```

2295     double result = 1./find_l(IGESKnot, a - 1, r);
2296     for (size_t i = a; i ≤ b; i++) {
2297         double tmp = find_l(IGESKnot, i, r);
2298         result *= (1. - tmp)/tmp;
2299     }
2300     return result;
2301 }
2302 double cubic_spline::find_l(const vector<double> &IGESKnot, long j, long r) {
2303     return (IGESKnot[r] - IGESKnot[r - 4 + j])/(IGESKnot[r + j] - IGESKnot[r - 4 + j]);
2304 }

```

183. <Methods of **cubic\_spline** 103> +≡

```

2305 protected:
2306     double get_blending_ratio(const vector<double> &, long, long, long);
2307     double bracket(const vector<double> &, long, long, long);
2308     double find_l(const vector<double> &, long, long);

```

184. Knot removal을 수행하는 method를 정의한다. 자세한 알고리즘은 Eck의 논문을 참조한다.

<Methods for knot insertion and removal of **cubic\_spline** 176> +≡

```

2309     void cubic_spline::remove_knot(const double u) {
2310         vector<double> IGESKnot;
2311         vector<point> forward;
2312         vector<point> backward;
2313         const int k = 4;
2314         <Set multiplicity of end knots to order of this curve instead of degree 185>;
2315         size_t r = find_index_in_knot_sequence(u) + 1;
2316         unsigned long v = find_multiplicity(u);
2317         <Determine forward control points 186>;
2318         <Determine backward control points 187>;
2319         <Blend forward and backward control points 188>;
2320         for (size_t i = r; i ≤ _knot_sqnc.size() - 1; i++) {
2321             _knot_sqnc[i - 1] = _knot_sqnc[i];
2322         }
2323         _knot_sqnc.pop_back();
2324     }

```

**185.** 〈Set multiplicity of end knots to order of this curve instead of degree 185〉  $\equiv$

```

2325  IGESKnot.push_back(_knot_sqnc[0]);
2326  for (size_t i = 0; i  $\neq$  _knot_sqnc.size(); ++i) {
2327      IGESKnot.push_back(_knot_sqnc[i]);
2328  }
2329  IGESKnot.push_back(_knot_sqnc.back());

```

이 코드는 184번 마디에서 사용된다.

**186.** 〈Determine forward control points 186〉  $\equiv$

```

2330  for (size_t i = 0; i  $\leq$  r - k + v - 1; i++) {
2331      forward.push_back(_ctrl_pts[i]);
2332  }
2333  for (size_t i = r - k + v; i  $\leq$  r - 1; i++) {
2334      double l = (IGESKnot[r] - IGESKnot[i]) / (IGESKnot[k + i] - IGESKnot[i]);
2335      forward.push_back(1.0/l * _ctrl_pts[i] + (1.0 - 1.0/l) * forward[i - 1]);
2336  }
2337  for (size_t i = r; i  $\leq$  _ctrl_pts.size() - 2; i++) {
2338      forward.push_back(_ctrl_pts[i + 1]);
2339  }

```

이 코드는 184번 마디에서 사용된다.

**187.** 〈Determine backward control points 187〉  $\equiv$

```

2340  for (size_t i = 0; i  $\leq$  _ctrl_pts.size() - 2; i++) {
2341      backward.push_back(cagd::point(2));
2342  }
2343  for (long i = _ctrl_pts.size() - 2; i  $\geq$  r - 1; i--) {
2344      backward[i] = _ctrl_pts[i + 1];
2345  }
2346  for (long i = r - 2; i  $\geq$  r - k + v - 1; i--) {
2347      double l = (IGESKnot[r] - IGESKnot[i + 1]) / (IGESKnot[k + i + 1] - IGESKnot[i + 1]);
2348      backward[i] = 1. / (1. - l) * _ctrl_pts[i + 1] + (1. - 1. / (1. - l)) * backward[i + 1];
2349  }
2350  for (long i = r - k + v - 2; i  $\geq$  0; i--) {
2351      backward[i] = _ctrl_pts[i];
2352  }

```

이 코드는 184번 마디에서 사용된다.

**188.** 〈Blend forward and backward control points 188〉  $\equiv$

```

2353   for (size_t  $i = r - k + v - 1$ ;  $i \leq r - 1$ ;  $i++$ ) {
2354       double  $\mu = \text{get\_blending\_ratio}(\text{IGESKnot}, v, r, i)$ ;
2355        $\_ctrl\_pts[i] = (1. - \mu) * \text{forward}[i] + \mu * \text{backward}[i]$ ;
2356   }
2357   for (size_t  $i = r$ ;  $i \leq \_ctrl\_pts.size() - 2$ ;  $i++$ ) {
2358        $\_ctrl\_pts[i] = \_ctrl\_pts[i + 1]$ ;
2359   }
2360    $\_ctrl\_pts.pop\_back()$ ;

```

이 코드는 184번 마디에서 사용된다.

**189.** 〈Methods of **cubic\_spline** 103〉  $+ \equiv$

```

2361   public:
2362       void  $\text{remove\_knot}(\text{const double})$ ;

```



**190.** PostScript 파일 출력을 위한 함수들은 다음과 같다. 곡선을 계산할 때 입력받는 변수 *dense*는 곡선을 몇 개의 선분 조각으로 근사화할 것인지 나타내므로 실제 계산해야 하는 곡선상의 점들은 그것보다 하나 더 많다.

〈Methods for PostScript output of **cubic\_spline** 190〉 ≡

```

2363 void cubic_spline::write_curve_in_postscript(
2364     psf &ps_file,
2365     unsigned dense,
2366     float line_width,
2367     int x, int y,
2368     float magnification
2369 ) const {
2370     ios_base::fmtflags previous_options = ps_file.flags();
2371     ps_file.precision(4);
2372     ps_file.setf(ios_base::fixed, ios_base::floatfield);
2373     ps_file << "newpath" << endl << "[ ]0 setdash" << line_width << " setlinewidth" << endl;
2374     point pt(magnification * evaluate(_knot_sqnc[2], 2));
2375     ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
2376     double incr = (_knot_sqnc[_knot_sqnc.size() - 3] - _knot_sqnc[2])/double(dense);
2377     for (size_t i = 0; i < dense + 1; i++) {
2378         double u = _knot_sqnc[2] + incr * i;
2379         pt = magnification * evaluate(u);
2380         ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
2381     }
2382 #if 0
2383     for (size_t i = 2; i < _knot_sqnc.size() - 3; i++) {
2384         if (_knot_sqnc[i] < _knot_sqnc[i + 1]) {
2385             double knot = _knot_sqnc[i];
2386             double incr = (_knot_sqnc[i + 1] - knot)/double(dense);
2387             double u = knot;
2388             for (size_t j = 0; j <= dense; j++) {
2389                 pt = magnification * evaluate(u, i);
2390                 ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
2391                 u += incr;
2392             }
2393         }
2394     }
2395 #endif
2396     ps_file << "stroke" << endl;
2397     ps_file.flags(previous_options);
2398 }
2399 void cubic_spline::write_control_polygon_in_postscript(

```

```

2400         psf &ps_file,          float line_width,
2401         int x, int y,
2402         float magnification
2403     ) const {
2404         ios_base::fmtflags previous_options = ps_file.flags();
2405         ps_file.precision(4);
2406         ps_file.setf(ios_base::fixed, ios_base::floatfield);
2407         ps_file << "newpath" << endl << "[]0setdash" << .5 * line_width << "setlinewidth" << endl;
2408         point pt(magnification * _ctrl_pts[0]);
2409         ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
2410         for (size_t i = 1; i < _ctrl_pts.size(); i++) {
2411             pt = magnification * _ctrl_pts[i];
2412             ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
2413         }
2414         ps_file << "stroke" << endl;
2415         ps_file.flags(previous_options);
2416     }

2417 void cubic_spline::write_control_points_in_postscript(
2418     psf &ps_file,
2419     float line_width,
2420     int x, int y,
2421     float magnification
2422 ) const {
2423     ios_base::fmtflags previous_options = ps_file.flags();
2424     ps_file.precision(4);
2425     ps_file.setf(ios_base::fixed, ios_base::floatfield);
2426     point pt(magnification * _ctrl_pts[0]);
2427     ps_file << "0setgray" << endl << "newpath" << endl << pt(x) << "\t" << pt(y) << "\t" <<
        (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl << "closepath" <<
        endl << "fillstroke" << endl;
2428     if (_ctrl_pts.size() > 2) {
2429         for (size_t i = 1; i ≤ (_ctrl_pts.size() - 2); i++) {
2430             pt = magnification * _ctrl_pts[i];
2431             ps_file << "newpath" << endl << pt(x) << "\t" << pt(y) << "\t" << (line_width * 3) << "\t" <<
                0.0 << "\t" << 360 << "\t" << "arc" << endl << "closepath" << endl << line_width <<
                "\t" << "setlinewidth" << endl << "stroke" << endl;
2432         }
2433         pt = magnification * _ctrl_pts.back();
2434         ps_file << "0setgray" << endl << "newpath" << endl << pt(x) << "\t" << pt(y) << "\t" <<
            (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl << "closepath" <<
            endl << "fillstroke" << endl;

```

```

2435     }
2436     ps_file.flags(previous_options);
2437     }

```

이 코드는 101번 마디에서 사용된다.

**191.** 〈Methods of **cubic\_spline** 103〉 +≡

```

2438 public:
2439     void write_curve_in_postscript(
2440         psf &, unsigned, float, int x = 1, int y = 1,
2441         float magnification = 1.0) const;
2442     void write_control_polygon_in_postscript(
2443         psf &, float, int x = 1, int y = 1,
2444         float magnification = 1.0) const;
2445     void write_control_points_in_postscript(
2446         psf &, float, int x = 1, int y = 1,
2447         float magnification = 1.0) const;

```

**192. Index.** 이 프로그램에 사용된 심볼과 그에 대한 설명을 보려면 아래의 인덱스를 참조하라.

- \_\_COMPUTER\_AIDED\_GEOMETRIC\_DESIGN\_H\_: [2](#).  
 \_ctrl\_pts: [27](#), [29](#), [31](#), [36](#), [38](#), [40](#), [43](#), [47](#), [52](#), [54](#), [59](#),  
[61](#), [64](#), [69](#), [72](#), [104](#), [110](#), [114](#), [120](#), [128](#), [144](#), [171](#),  
[172](#), [176](#), [178](#), [179](#), [180](#), [182](#), [186](#), [187](#), [188](#), [190](#).  
 \_curves: [74](#), [76](#), [78](#), [80](#), [82](#), [84](#), [86](#), [88](#).  
 \_degree: [43](#), [45](#), [47](#), [49](#), [52](#), [54](#), [59](#), [60](#), [61](#), [62](#), [64](#), [69](#).  
 \_elem: [8](#), [10](#), [12](#), [14](#), [16](#), [18](#), [20](#), [22](#).  
 \_err: [27](#), [40](#), [64](#), [69](#), [120](#), [148](#), [155](#), [171](#), [176](#).  
 \_interpolate: [144](#), [145](#), [146](#).  
 \_kernel\_id: [100](#), [102](#), [106](#), [114](#), [146](#).  
 \_knot\_sqnc: [100](#), [102](#), [104](#), [106](#), [108](#), [110](#), [112](#), [114](#),  
[116](#), [120](#), [122](#), [124](#), [126](#), [144](#), [150](#), [151](#), [152](#), [153](#),  
[170](#), [176](#), [179](#), [180](#), [184](#), [185](#), [190](#).  
 \_mp: [100](#), [102](#), [106](#), [112](#), [114](#), [146](#).  
 \_WIN32: [2](#).  
 \_WIN64: [2](#).  
 a: [154](#), [182](#).  
 Ainv: [135](#).  
 alpha: [130](#), [132](#), [137](#), [138](#), [139](#), [140](#), [142](#).  
 alpha\_i: [154](#).  
 area: [56](#).  
 argc: [6](#).  
 argv: [6](#).  
 at: [118](#).  
 B: [137](#).  
 b: [54](#), [55](#), [135](#), [137](#), [142](#), [154](#), [182](#).  
 back: [69](#), [72](#), [116](#), [120](#), [126](#), [146](#), [170](#), [185](#), [190](#).  
 backup\_point: [64](#).  
 backup1: [69](#).  
 backup2: [69](#).  
 backward: [184](#), [187](#), [188](#).  
 begin: [12](#), [29](#), [36](#), [47](#), [78](#), [84](#), [88](#), [120](#), [122](#),  
[146](#), [151](#), [176](#).  
 bessell: [143](#).  
 beta: [130](#), [132](#), [137](#), [138](#), [139](#), [142](#), [182](#).  
 beta\_i: [154](#).  
 bezier: [1](#), [42](#), [43](#), [44](#), [45](#), [47](#), [48](#), [49](#), [50](#), [52](#), [54](#),  
[58](#), [63](#), [64](#), [69](#), [72](#), [74](#), [80](#), [81](#), [86](#), [90](#), [91](#), [92](#),  
[93](#), [94](#), [95](#), [118](#), [174](#).  
 bezier\_control\_points: [118](#), [168](#), [173](#), [174](#).  
 bezier\_ctrl\_points: [168](#), [172](#), [174](#).  
 bezier\_ctrlpt: [118](#).  
 bezier\_curve: [118](#).  
 bhat\_fxn: [140](#).  
 bracket: [182](#), [183](#).  
 brk: [182](#).  
 buffer: [22](#), [38](#), [108](#).  
 buffer\_property: [112](#), [114](#).  
 c: [154](#).  
 c\_str: [4](#).  
 cagd: [2](#), [3](#), [4](#), [6](#), [16](#), [24](#), [27](#), [54](#), [56](#), [67](#), [69](#), [86](#), [120](#),  
[130](#), [132](#), [133](#), [135](#), [137](#), [151](#), [152](#), [187](#).  
 centripetal: [143](#), [148](#), [166](#).  
 chord\_length: [143](#), [147](#), [148](#).  
 chrono: [6](#).  
 clamped: [143](#), [146](#), [155](#).  
 clear: [47](#), [59](#), [61](#), [69](#), [91](#), [92](#), [93](#), [94](#), [95](#), [126](#),  
[128](#), [144](#), [168](#), [174](#), [176](#).  
 close: [4](#).  
 close\_postscript\_file: [4](#), [5](#), [90](#), [159](#), [164](#), [166](#).  
 coeff: [52](#).  
 combi: [69](#).  
 cond: [144](#), [146](#), [147](#), [155](#).  
 const\_ctrlpt\_itr: [27](#).  
 const\_curve\_itr: [74](#), [78](#), [88](#).  
 const\_iterator: [27](#), [36](#), [47](#), [74](#), [100](#).  
 const\_knot\_itr: [100](#), [122](#), [123](#).  
 constant: [115](#).  
 control\_points: [104](#), [105](#).  
 convert\_int: [115](#).  
 cos: [166](#).  
 count: [78](#), [79](#), [159](#), [166](#).  
 counter: [64](#), [69](#).  
 cout: [6](#), [26](#), [132](#), [142](#), [159](#), [161](#), [166](#).  
 cp: [114](#).  
 cp\_buffer: [114](#).  
 cpts: [115](#), [174](#).  
 create\_buffer: [112](#), [114](#).  
 create\_kernel: [102](#), [146](#).  
 create\_postscript\_file: [4](#), [5](#), [90](#), [159](#), [164](#), [166](#).  
 crv: [40](#), [78](#), [80](#), [82](#), [84](#), [88](#), [106](#), [112](#), [115](#), [159](#),  
[161](#), [164](#), [166](#).

- crv\_pts\_p*: [159](#), [166](#).
- crv\_pts\_s*: [159](#), [166](#).
- ctrl\_pts*: [31](#), [32](#), [88](#), [90](#), [91](#), [92](#), [93](#), [94](#), [95](#).
- ctrl\_pts\_size*: [31](#), [32](#), [88](#).
- ctrlpt\_itr**: [27](#).
- cubic\_spline**: [1](#), [99](#), [100](#), [101](#), [102](#), [103](#), [104](#), [106](#),  
[107](#), [108](#), [110](#), [112](#), [116](#), [118](#), [122](#), [124](#), [126](#), [128](#),  
[143](#), [144](#), [146](#), [147](#), [148](#), [155](#), [159](#), [164](#), [166](#),  
[168](#), [174](#), [176](#), [182](#), [184](#), [190](#).
- curvature*: [174](#).
- curvature\_at\_zero*: [54](#), [55](#).
- curve**: [1](#), [27](#), [28](#), [29](#), [31](#), [36](#), [37](#), [38](#), [40](#), [41](#), [42](#), [43](#),  
[45](#), [49](#), [74](#), [76](#), [82](#), [99](#), [100](#), [102](#), [106](#), [108](#), [146](#).
- curve\_itr**: [74](#), [84](#).
- curves*: [90](#), [91](#), [92](#), [93](#), [94](#), [95](#).
- d*: [115](#), [135](#), [138](#), [155](#).
- d\_minus*: [155](#).
- d\_plus*: [155](#).
- datum*: [160](#).
- deg*: [90](#).
- degree*: [30](#), [45](#), [46](#), [78](#), [79](#), [90](#), [104](#), [105](#).
- DEGREE\_ELEVATION\_FAIL: [64](#), [66](#).
- DEGREE\_MISMATCH: [51](#).
- DEGREE\_REDUCTION\_FAIL: [69](#), [71](#).
- delta*: [54](#), [118](#), [124](#), [125](#), [151](#), [152](#), [154](#), [155](#),  
[157](#), [158](#), [172](#).
- delta\_i*: [154](#).
- delta\_im1*: [154](#).
- delta\_im2*: [154](#).
- delta\_ip1*: [154](#).
- dense*: [190](#).
- density*: [54](#), [174](#).
- derivative*: [35](#), [52](#), [53](#), [86](#), [87](#), [118](#), [119](#).
- description*: [22](#), [23](#), [38](#), [39](#), [108](#), [109](#), [159](#), [166](#).
- dgr*: [64](#), [69](#), [78](#), [84](#).
- diff*: [159](#).
- dim*: [10](#), [11](#), [14](#), [16](#), [20](#), [22](#), [26](#), [29](#), [30](#), [36](#), [38](#), [78](#),  
[79](#), [112](#), [120](#), [135](#), [137](#), [155](#).
- dimension*: [10](#), [11](#), [29](#), [30](#), [78](#), [79](#), [86](#).
- dist*: [20](#), [21](#), [24](#), [25](#), [26](#), [54](#), [151](#), [152](#), [159](#), [166](#).
- drv*: [118](#).
- du*: [115](#), [159](#), [166](#).
- duration\_cast**: [159](#), [166](#).
- e*: [54](#), [55](#), [146](#).
- Env*: [137](#), [138](#), [139](#), [140](#).
- elem*: [130](#).
- elevate\_degree*: [64](#), [65](#), [84](#), [85](#), [90](#).
- Enb*: [139](#).
- end*: [12](#), [47](#), [78](#), [84](#), [88](#), [122](#), [144](#), [151](#), [154](#).
- end\_condition**: [143](#), [144](#), [145](#), [146](#), [147](#), [155](#),  
[159](#), [166](#).
- endl*: [4](#), [6](#), [22](#), [38](#), [72](#), [88](#), [108](#), [132](#), [142](#), [159](#),  
[161](#), [166](#), [190](#).
- enqueue\_data\_parallel\_kernel*: [114](#).
- enqueue\_read\_buffer*: [112](#).
- enqueue\_write\_buffer*: [114](#).
- EPS: [2](#).
- err*: [161](#).
- err\_code**: [2](#), [27](#).
- evaluate*: [35](#), [52](#), [53](#), [72](#), [86](#), [87](#), [88](#), [110](#), [111](#),  
[159](#), [161](#), [166](#), [190](#).
- evaluate\_all*: [112](#), [113](#), [159](#), [166](#).
- evaluate\_crv*: [115](#).
- exit*: [4](#).
- E1b*: [139](#).
- fabs*: [54](#).
- factorial*: [67](#), [68](#), [69](#).
- file*: [90](#), [159](#), [164](#), [166](#).
- file\_name*: [4](#).
- fill*: [6](#).
- find*: [122](#).
- find\_index\_in\_knot\_sequence*: [110](#), [116](#), [117](#), [176](#),  
[184](#).
- find\_index\_in\_sequence*: [116](#), [117](#), [118](#).
- find\_l*: [182](#), [183](#).
- find\_multiplicity*: [122](#), [123](#), [184](#).
- fixed*: [72](#), [88](#), [190](#).
- flags*: [72](#), [88](#), [190](#).
- floatfield*: [72](#), [88](#), [190](#).
- floor*: [86](#).
- fmtflags*: [72](#), [88](#), [190](#).
- forward*: [184](#), [186](#), [188](#).
- front*: [69](#), [120](#).
- function\_spline*: [143](#), [148](#), [159](#).

*gamma*: [130](#), [132](#), [137](#), [138](#), [139](#), [140](#), [142](#), [182](#).

*gamma\_i*: [154](#).

*get\_blending\_ratio*: [182](#), [183](#), [188](#).

*get\_global\_id*: [115](#).

**global**: [115](#).

*h*: [54](#).

*half*: [54](#).

*head*: [128](#).

**high\_resolution\_clock**: [159](#), [166](#).

*I*: [110](#), [115](#).

*i*: [12](#), [14](#), [16](#), [18](#), [20](#), [22](#), [26](#), [31](#), [36](#), [38](#), [47](#), [52](#), [54](#),  
[59](#), [60](#), [61](#), [62](#), [64](#), [69](#), [72](#), [88](#), [108](#), [110](#), [112](#),  
[114](#), [115](#), [116](#), [118](#), [124](#), [126](#), [128](#), [130](#), [132](#), [133](#),  
[135](#), [137](#), [142](#), [146](#), [150](#), [151](#), [152](#), [153](#), [154](#), [155](#),  
[158](#), [159](#), [160](#), [161](#), [166](#), [170](#), [172](#), [174](#), [178](#), [179](#),  
[180](#), [182](#), [184](#), [185](#), [186](#), [187](#), [188](#), [190](#).

*id*: [115](#).

*IGESKnot*: [182](#), [184](#), [185](#), [186](#), [187](#), [188](#).

*incr*: [190](#).

*index*: [61](#), [86](#), [118](#), [176](#), [178](#), [179](#), [180](#).

*initial*: [144](#), [154](#).

**initializer\_list**: [12](#), [13](#).

*insert*: [176](#).

*insert\_end\_knots*: [126](#), [127](#), [144](#).

*insert\_knot*: [176](#), [181](#).

*intermediate*: [128](#).

*interpolate*: [144](#).

*interpolated*: [161](#).

*inv*: [132](#).

*inverse*: [130](#).

*invert\_tridiagonal*: [130](#), [131](#), [132](#), [135](#), [138](#).

**ios\_base**: [4](#), [72](#), [88](#), [190](#).

*iter*: [122](#).

**iterator**: [27](#), [74](#), [100](#).

*j*: [69](#), [112](#), [114](#), [115](#), [130](#), [132](#), [137](#), [138](#), [139](#),  
[140](#), [174](#), [182](#), [190](#).

*k*: [110](#), [115](#), [130](#), [133](#), [135](#), [184](#).

*kappa*: [54](#), [174](#).

**kernel**: [115](#).

*knot*: [168](#), [170](#), [171](#), [172](#), [174](#), [190](#).

**knot\_itr**: [100](#), [151](#).

*knot\_sequence*: [104](#), [105](#), [159](#), [166](#).

*knots*: [102](#), [114](#), [115](#), [118](#), [159](#), [166](#).

*knots\_buffer*: [114](#).

*L*: [112](#), [115](#), [154](#), [172](#).

*l*: [135](#), [138](#), [186](#), [187](#).

*lambda*: [69](#).

*left*: [54](#), [58](#), [61](#).

*line\_width*: [72](#), [88](#), [190](#).

**list**: [36](#), [37](#), [47](#), [48](#), [174](#).

*l2r*: [69](#).

*m*: [112](#).

**M\_PI**: [2](#), [160](#), [161](#), [166](#).

**M\_PI\_2**: [2](#).

**M\_PI\_4**: [2](#).

*magnification*: [33](#), [72](#), [73](#), [88](#), [89](#), [190](#), [191](#).

*main*: [6](#).

*mat*: [133](#).

*matlab\_bench*: [161](#).

**MAX\_BUFF\_SIZE**: [115](#).

*max\_u*: [86](#).

**milliseconds**: [159](#), [166](#).

*min*: [14](#).

**mpoi**: [2](#), [100](#), [112](#), [114](#).

*mu*: [188](#).

*multiply*: [133](#), [134](#), [135](#), [140](#).

*mv*: [133](#).

*m1*: [182](#).

*m2*: [182](#).

*N*: [112](#), [115](#).

*n*: [12](#), [20](#), [67](#), [110](#), [112](#), [115](#), [128](#), [130](#), [133](#),  
[135](#), [137](#), [176](#).

*natural*: [143](#).

*negated*: [16](#).

*new\_ctrl\_pts*: [176](#), [178](#), [179](#), [180](#).

*newKnots*: [126](#).

**NO\_ERR**: [2](#).

**NOMINMAX**: [2](#).

*not\_a\_knot*: [143](#), [147](#), [155](#), [159](#).

**NOT\_INSERTABLE\_KNOT**: [176](#), [177](#).

*now*: [159](#), [166](#).

*num\_ctrlpts*: [114](#).

*num\_knots*: [114](#).

**ofstream**: [2](#).

- one\_third*: [146](#).  
*open*: [4](#).  
*out*: [4](#).  
 OUT\_OF\_KNOT\_RANGE: [120](#), [121](#), [176](#).  
 OUTPUT\_FILE\_OPEN\_FAIL: [34](#).  
*p*: [144](#), [146](#), [160](#), [164](#), [166](#).  
**parametrization**: [143](#), [144](#), [145](#), [146](#), [147](#),  
     [148](#), [159](#), [166](#).  
*periodic*: [143](#), [155](#), [166](#).  
*phi*: [130](#).  
**piecewise\_bezier\_curve**: [1](#), [74](#), [75](#), [76](#), [77](#), [78](#),  
     [80](#), [82](#), [83](#), [84](#), [86](#), [88](#), [90](#).  
**point**: [1](#), [7](#), [8](#), [9](#), [10](#), [12](#), [13](#), [14](#), [15](#), [16](#), [17](#), [18](#),  
     [20](#), [21](#), [22](#), [24](#), [25](#), [26](#), [27](#), [31](#), [32](#), [35](#), [36](#), [37](#),  
     [47](#), [48](#), [52](#), [53](#), [54](#), [55](#), [56](#), [57](#), [58](#), [64](#), [69](#), [72](#),  
     [86](#), [87](#), [88](#), [90](#), [91](#), [92](#), [93](#), [94](#), [95](#), [102](#), [103](#), [104](#),  
     [105](#), [110](#), [111](#), [112](#), [113](#), [118](#), [119](#), [120](#), [128](#), [129](#),  
     [135](#), [136](#), [137](#), [141](#), [142](#), [144](#), [145](#), [146](#), [147](#),  
     [154](#), [155](#), [159](#), [160](#), [164](#), [166](#), [168](#), [172](#), [173](#),  
     [174](#), [175](#), [176](#), [184](#), [187](#), [190](#).  
*points*: [47](#), [58](#), [59](#), [60](#), [61](#), [62](#).  
*pop\_back*: [69](#), [158](#), [184](#), [188](#).  
*pow*: [69](#), [130](#).  
*precision*: [72](#), [88](#), [190](#).  
*prev*: [6](#).  
*previous\_options*: [72](#), [88](#), [190](#).  
*print\_title*: [6](#), [26](#), [90](#), [132](#), [142](#), [159](#), [164](#), [166](#).  
*prod*: [130](#).  
*ps\_file*: [4](#), [72](#), [88](#), [190](#).  
**psf**: [2](#), [4](#), [5](#), [33](#), [72](#), [73](#), [88](#), [89](#), [90](#), [159](#), [164](#),  
     [166](#), [190](#), [191](#).  
*pt*: [14](#), [16](#), [20](#), [36](#), [72](#), [88](#), [112](#), [190](#).  
*pts*: [36](#), [102](#), [112](#).  
*pts\_buffer*: [112](#), [114](#).  
*pt1*: [16](#), [24](#).  
*pt2*: [16](#), [24](#).  
*push\_back*: [47](#), [52](#), [54](#), [59](#), [61](#), [64](#), [69](#), [80](#), [81](#), [91](#),  
     [92](#), [93](#), [94](#), [95](#), [110](#), [118](#), [126](#), [144](#), [150](#), [151](#),  
     [152](#), [153](#), [154](#), [160](#), [164](#), [166](#), [170](#), [172](#), [174](#),  
     [178](#), [179](#), [180](#), [185](#), [186](#), [187](#).  
*p0*: [26](#).  
*p1*: [26](#), [56](#).  
*p2*: [26](#), [56](#).  
*p3*: [26](#), [56](#).  
*quadratic*: [143](#).  
*r*: [52](#), [60](#), [62](#), [76](#), [135](#), [154](#), [166](#), [182](#), [184](#).  
*r\_f*: [157](#).  
*r\_i*: [157](#).  
*ratio*: [64](#).  
 READ\_ONLY: [114](#).  
 READ\_WRITE: [112](#).  
*reduce\_degree*: [69](#), [70](#), [84](#), [85](#), [90](#).  
*release\_buffer*: [112](#).  
*remove\_knot*: [184](#), [189](#).  
*result*: [182](#).  
*right*: [54](#), [58](#), [59](#).  
*r2l*: [69](#).  
*r2l\_reversed*: [69](#).  
*r2l\_reversed\_size*: [69](#).  
*s*: [14](#), [16](#).  
*s\_f*: [157](#).  
*s\_i*: [157](#).  
*scheme*: [144](#), [146](#), [147](#), [148](#).  
*segment*: [174](#).  
*set\_control\_points*: [128](#), [129](#), [155](#).  
*set\_kernel\_argument*: [114](#).  
*setf*: [72](#), [88](#), [190](#).  
*setw*: [6](#).  
*shifter*: [110](#), [115](#).  
*sign*: [115](#).  
*signed\_area*: [54](#), [56](#), [57](#).  
*signed\_curvature*: [54](#), [55](#), [174](#), [175](#).  
*sin*: [160](#), [166](#).  
*size*: [10](#), [18](#), [29](#), [31](#), [36](#), [38](#), [47](#), [52](#), [59](#), [61](#), [64](#), [69](#),  
     [72](#), [78](#), [86](#), [108](#), [112](#), [114](#), [116](#), [124](#), [126](#), [128](#),  
     [130](#), [133](#), [135](#), [137](#), [144](#), [150](#), [151](#), [152](#), [153](#), [154](#),  
     [159](#), [166](#), [170](#), [171](#), [172](#), [174](#), [176](#), [180](#), [182](#),  
     [184](#), [185](#), [186](#), [187](#), [188](#), [190](#).  
 SIZE\_MAX: [116](#), [176](#).  
*solve\_cyclic\_tridiagonal\_system*: [137](#), [141](#), [142](#), [155](#).  
*solve\_tridiagonal\_system*: [135](#), [136](#), [155](#).  
*spline*: [161](#), [162](#).  
*splines*: [118](#).  
*sqnc*: [116](#).

- sqrt*: [20](#), [152](#), [161](#).  
*src*: [12](#), [14](#), [36](#), [47](#), [49](#), [102](#).  
**std**: [2](#), [6](#), [20](#), [54](#), [69](#), [86](#).  
*step*: [72](#), [88](#).  
*steps*: [159](#), [166](#).  
*str*: [6](#), [22](#), [38](#), [108](#).  
**string**: [4](#), [5](#), [22](#), [23](#), [38](#), [39](#), [108](#), [109](#).  
**stringstream**: [22](#), [38](#), [108](#).  
*subdivision*: [54](#), [58](#), [63](#).  
*sum*: [20](#).  
*sum\_delta*: [151](#), [152](#).  
*sz*: [14](#), [16](#).  
*sz\_min*: [14](#).  
*t*: [52](#), [54](#), [58](#), [72](#), [88](#), [115](#), [118](#).  
*tail*: [128](#).  
*theta*: [130](#).  
*tmp*: [110](#), [115](#), [182](#).  
*tmp\_point*: [64](#).  
**TRIDIAGONAL\_NOT\_SOLVABLE**: [155](#), [156](#).  
*true*: [4](#), [90](#), [159](#), [164](#), [166](#).  
*two\_third*: [146](#).  
*t0*: [159](#), [166](#).  
*t1*: [52](#), [58](#), [60](#), [61](#), [62](#), [110](#), [159](#), [166](#).  
*t2*: [110](#).  
*u*: [86](#), [110](#), [115](#), [116](#), [118](#), [122](#), [135](#), [138](#), [161](#),  
[176](#), [184](#), [190](#).  
**UNABLE\_TO\_BREAK\_INTO\_BEZIER**: [169](#), [171](#).  
*uniform*: [143](#), [148](#).  
**UNKNOWN\_END\_CONDITION**: [155](#), [156](#).  
**UNKNOWN\_PARAMETRIZATION**: [148](#), [149](#).  
*us*: [159](#), [166](#).  
*v*: [12](#), [182](#), [184](#).  
*vec*: [133](#).  
**vector**: [8](#), [12](#), [27](#), [36](#), [37](#), [47](#), [48](#), [52](#), [54](#), [55](#), [58](#),  
[69](#), [74](#), [90](#), [100](#), [102](#), [103](#), [104](#), [105](#), [110](#), [112](#),  
[113](#), [114](#), [116](#), [117](#), [118](#), [126](#), [128](#), [129](#), [130](#), [131](#),  
[132](#), [133](#), [134](#), [135](#), [136](#), [137](#), [138](#), [140](#), [141](#), [142](#),  
[144](#), [145](#), [146](#), [147](#), [154](#), [155](#), [159](#), [160](#), [164](#), [166](#),  
[168](#), [173](#), [174](#), [175](#), [176](#), [182](#), [183](#), [184](#).  
*v1*: [12](#).  
*v2*: [12](#).  
*v3*: [12](#), [13](#).  
*with\_new\_page*: [4](#).  
*write\_control\_points\_in\_postscript*: [33](#), [72](#), [73](#), [88](#),  
[89](#), [90](#), [159](#), [164](#), [166](#), [190](#), [191](#).  
*write\_control\_polygon\_in\_postscript*: [33](#), [72](#), [73](#), [88](#),  
[89](#), [90](#), [159](#), [164](#), [166](#), [190](#), [191](#).  
*write\_curve\_in\_postscript*: [33](#), [72](#), [73](#), [88](#), [89](#), [90](#),  
[159](#), [164](#), [166](#), [190](#), [191](#).  
*x*: [33](#), [72](#), [73](#), [88](#), [89](#), [135](#), [137](#), [142](#), [155](#), [190](#), [191](#).  
*x\_n*: [137](#), [139](#), [140](#).  
*x\_n\_den*: [139](#).  
*x\_n\_num*: [139](#).  
*xhat*: [137](#), [140](#).  
*xi*: [135](#).  
*y*: [33](#), [72](#), [73](#), [88](#), [89](#), [161](#), [190](#), [191](#).



- 〈Access control points of **curve** 31〉 28번 마디에서 사용된다.
- 〈Blend forward and backward control points 188〉 184번 마디에서 사용된다.
- 〈Build-up 1st brush 93〉 90번 마디에서 사용된다.
- 〈Build-up 2nd, 4th, and 5th brush 92〉 90번 마디에서 사용된다.
- 〈Build-up 3rd brush 91〉 90번 마디에서 사용된다.
- 〈Build-up 6th, 7th, 8th, and 9th brush (inner part) 95〉 90번 마디에서 사용된다.
- 〈Build-up 6th, 7th, 8th, and 9th brush (outer part) 94〉 90번 마디에서 사용된다.
- 〈Calculate  $E^{-1}$  138〉 137번 마디에서 사용된다.
- 〈Calculate  $\hat{\mathbf{x}}$  140〉 137번 마디에서 사용된다.
- 〈Calculate  $x_n$  139〉 137번 마디에서 사용된다.
- 〈Calculate Bézier control points 172〉 168번 마디에서 사용된다.
- 〈Calculate points on a cubic spline using OpenCL Kernel 114〉 112번 마디에서 사용된다.
- 〈Centripetal parametrization of knot sequence 152〉 148번 마디에서 사용된다.
- 〈Check the range of knot value given 120〉 118번 마디에서 사용된다.
- 〈Check whether the curve can be broken into Bézier curves 171〉 168번 마디에서 사용된다.
- 〈Chord length parametrization of knot sequence 151〉 148번 마디에서 사용된다.
- 〈Compare the result of interpolation with MATLAB 161〉 159번 마디에서 사용된다.
- 〈Construct new control points by piecewise linear interpolation 179〉 176번 마디에서 사용된다.
- 〈Constructor and destructor of **curve** 36〉 28번 마디에서 사용된다.
- 〈Constructors and destructor of **bezier** 47〉 44번 마디에서 사용된다.
- 〈Constructors and destructor of **cubic\_spline** 102, 146〉 101번 마디에서 사용된다.
- 〈Constructors and destructor of **piecewise\_bezier\_curve** 76〉 75번 마디에서 사용된다.
- 〈Constructors and destructor of **point** 12〉 9번 마디에서 사용된다.
- 〈Copy control points for  $i = 0, \dots, I - d + 1$  178〉 176번 마디에서 사용된다.
- 〈Copy remaining control points to new control points 180〉 176번 마디에서 사용된다.
- 〈Create a new knot sequence of which each knot has multiplicity of 1 170〉 168번 마디에서 사용된다.
- 〈Data members of **bezier** 43〉 42번 마디에서 사용된다.
- 〈Data members of **cubic\_spline** 100〉 99번 마디에서 사용된다.
- 〈Data members of **point** 8〉 7번 마디에서 사용된다.
- 〈Declaration of **cagd** functions 5, 17, 25, 57, 68, 131, 134, 136, 141〉 2번 마디에서 사용된다.
- 〈Definition of **bezier** 42〉 2번 마디에서 사용된다.
- 〈Definition of **cubic\_spline** 99〉 2번 마디에서 사용된다.
- 〈Definition of **curve** 27〉 2번 마디에서 사용된다.
- 〈Definition of **piecewise\_bezier\_curve** 74〉 2번 마디에서 사용된다.
- 〈Definition of **point** 7〉 2번 마디에서 사용된다.
- 〈Degree elevation and reduction of **bezier** 64, 69〉 44번 마디에서 사용된다.
- 〈Degree elevation and reduction of **piecewise\_bezier\_curve** 84〉 75번 마디에서 사용된다.
- 〈Description of **cubic\_spline** 108〉 101번 마디에서 사용된다.
- 〈Determine backward control points 187〉 184번 마디에서 사용된다.
- 〈Determine forward control points 186〉 184번 마디에서 사용된다.
- 〈Enumerations of **cubic\_spline** 143〉 99번 마디에서 사용된다.
- 〈Error codes of **cagd** 34, 51, 66, 71, 121, 149, 156, 169, 177〉 2번 마디에서 사용된다.

- 〈Evaluation and derivative of **cubic\_spline** 110, 112, 118〉 101번 마디에서 사용된다.
- 〈Evaluation and derivative of **piecewise\_bezier\_curve** 86〉 75번 마디에서 사용된다.
- 〈Evaluation of **bezier** 52, 54〉 44번 마디에서 사용된다.
- 〈Function spline parametrization of knot sequence 153〉 148번 마디에서 사용된다.
- 〈Generate example data points 160〉 159번 마디에서 사용된다.
- 〈Generate knot sequence according to given parametrization scheme 148〉 144번 마디에서 사용된다.
- 〈Implementation of **bezier** 44〉 3번 마디에서 사용된다.
- 〈Implementation of **cagd** functions 4, 56, 67, 130, 133, 135, 137〉 3번 마디에서 사용된다.
- 〈Implementation of **cubic\_spline** 101〉 3번 마디에서 사용된다.
- 〈Implementation of **curve** 28〉 3번 마디에서 사용된다.
- 〈Implementation of **piecewise\_bezier\_curve** 75〉 3번 마디에서 사용된다.
- 〈Implementation of **point** 9〉 3번 마디에서 사용된다.
- 〈Methods for PostScript output of **cubic\_spline** 190〉 101번 마디에서 사용된다.
- 〈Methods for debugging of **curve** 38〉 28번 마디에서 사용된다.
- 〈Methods for interpolation of **cubic\_spline** 144〉 101번 마디에서 사용된다.
- 〈Methods for knot insertion and removal of **cubic\_spline** 176, 184〉 101번 마디에서 사용된다.
- 〈Methods of **bezier** 46, 48, 50, 53, 55, 63, 65, 70, 73〉 42번 마디에서 사용된다.
- 〈Methods of **cubic\_spline** 103, 105, 107, 109, 111, 113, 117, 119, 123, 125, 127, 129, 145, 147, 173, 175, 181, 183, 189, 191〉 99번 마디에서 사용된다.
- 〈Methods of **curve** 30, 32, 33, 35, 37, 39, 41〉 27번 마디에서 사용된다.
- 〈Methods of **piecewise\_bezier\_curve** 77, 79, 81, 83, 85, 87, 89〉 74번 마디에서 사용된다.
- 〈Methods of **point** 11, 13, 15, 19, 21, 23〉 7번 마디에서 사용된다.
- 〈Methods to calculate curvature of **cubic\_spline** 174〉 101번 마디에서 사용된다.
- 〈Methods to obtain a **bezier** curve for a segment of **cubic\_spline** 168〉 101번 마디에서 사용된다.
- 〈Miscellaneous methods of **cubic\_spline** 116, 122, 124, 126, 128, 182〉 101번 마디에서 사용된다.
- 〈Modification of **piecewise\_bezier\_curve** 80〉 75번 마디에서 사용된다.
- 〈Modify equations according to end conditions and solve them 155〉 144번 마디에서 사용된다.
- 〈Modify equations according to not-a-knot end condition 157〉 155번 마디에서 사용된다.
- 〈Modify equations according to periodic end condition 158〉 155번 마디에서 사용된다.
- 〈Non-member functions for **point** 16, 24〉 9번 마디에서 사용된다.
- 〈Obtain the left subpolygon of Bézier curve 61〉 58번 마디에서 사용된다.
- 〈Obtain the left subpolygon using de Casteljau algorithm 62〉 61번 마디에서 사용된다.
- 〈Obtain the right subpolygon of Bézier curve 59〉 58번 마디에서 사용된다.
- 〈Obtain the right subpolygon using the de Casteljau algorithm 60〉 59번 마디에서 사용된다.
- 〈Operators of **bezier** 49〉 44번 마디에서 사용된다.
- 〈Operators of **cubic\_spline** 106〉 101번 마디에서 사용된다.
- 〈Operators of **curve** 40〉 28번 마디에서 사용된다.
- 〈Operators of **piecewise\_bezier\_curve** 82〉 75번 마디에서 사용된다.
- 〈Operators of **point** 14, 18〉 9번 마디에서 사용된다.
- 〈Other member functions of **point** 20, 22〉 9번 마디에서 사용된다.
- 〈Output to PostScript of **bezier** 72〉 44번 마디에서 사용된다.
- 〈PostScript output of **piecewise\_bezier\_curve** 88〉 75번 마디에서 사용된다.

- 〈Properties of **bezier** 45〉 44번 마디에서 사용된다.
- 〈Properties of **cubic\_spline** 104〉 101번 마디에서 사용된다.
- 〈Properties of **curve** 29〉 28번 마디에서 사용된다.
- 〈Properties of **piecewise\_bezier\_curve** 78〉 75번 마디에서 사용된다.
- 〈Properties of **point** 10〉 9번 마디에서 사용된다.
- 〈Set multiplicity of end knots to order of this curve instead of degree 185〉 184번 마디에서 사용된다.
- 〈Setup equations of cubic spline interpolation 154〉 144번 마디에서 사용된다.
- 〈Subdivision of **bezier** 58〉 44번 마디에서 사용된다.
- 〈Test routines 26, 90, 132, 142, 159, 164, 166〉 6번 마디에서 사용된다.
- 〈Uniform parametrization of knot sequence 150〉 148번 마디에서 사용된다.
- 〈**cagd.cpp** 3〉
- 〈**cagd.h** 2〉
- 〈**cspline.cl** 115〉
- 〈**test.cpp** 6〉

# Computer-Aided Geometric Design

(Last revised on October 28, 2016)

	마디	쪽
Introduction .....	1	1
Namespace .....	2	2
Test program .....	6	5
Point in Euclidean space .....	7	6
Generic Curve .....	27	14
Bézier Curve .....	42	19
Piecewise Bézier Curve .....	74	34
Cubic Spline Curve .....	99	59
Index .....	192	116

Copyright © 2015–2016 by Changmook Chun

This document is published by Changmook Chun. All rights reserved. No part of this publication may be reproduced, stored in a retrieval systems, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author.