October 28, 2016 00:28

1. Introduction. 이 문서는 Gerald Farin의 "Curves and Surfaces for CAGD: A Practical Guide" 4th edition에 기술되어 있는 알고리즘들을 간략하게 설명하고 구현한다.

가장 먼저 n-차원 유클리드 공간에 존재하는 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 ⟩ ≡

```
#ifndef __COMPUTER_AIDED_GEOMETRIC_DESIGN_H_
#define __COMPUTER_AIDED_GEOMETRIC_DESIGN_H_
#include <cstddef>
#include <vector>
#include <list>
#include <string>
#include <cstdarg>
#include <algorithm>
#include <cmath>
#include <iostream>
#include <ostream>
#include <fstream>
#include <sstream>
#include <ios>
#include <iterator>
#include <initializer_list>
#include "mpoi.h"
#if defined (_WIN32) ∨ defined (_WIN64)
#define NOMINMAX
#define M_PI  3.14159265358979323846
#define M_PI_2  1.57079632679489661923
#define M_PI_4  0.785398163397448309616
#endif
  using namespace std;
  namespace cagd {
    const double EPS = 2.2204 · 10^-16;
    enum err_code {
      ⟨ Error codes of cagd 34 ⟩
      NO_ERR
```

```
    };
    typedef ofstream psf;
    ⟨ Definition of point 7 ⟩
    ⟨ Definition of curve 27 ⟩
    ⟨ Definition of bezier 42 ⟩
    ⟨ Definition of piecewise_bezier_curve 74 ⟩
    ⟨ Definition of cubic_spline 99 ⟩
    ⟨ Declaration of cagd functions 5 ⟩
  }
#endif
```

3. Implementation of **cagd**.

⟨ cagd.cpp 3 ⟩ ≡

```
#include "cagd.h"
  using namespace cagd;
  ⟨ Implementation of cagd functions 4 ⟩
  ⟨ Implementation of point 9 ⟩
  ⟨ Implementation of curve 28 ⟩
  ⟨ Implementation of bezier 44 ⟩
  ⟨ Implementation of piecewise_bezier_curve 75 ⟩
  ⟨ Implementation of cubic_spline 101 ⟩
```

4. PostScript 파일을 생성하고 닫기 위한 함수를 정의한다.

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

```
  psf cagd::create_postscript_file(string file_name) {
    psf ps_file;
    ps_file.open(file_name.c_str(), ios_base::out);
    if (¬ps_file) {
      exit(−1);
    }
    ps_file ≪ "%!PS-Adobe-3.0" ≪ endl ≪ "/Helvetica␣findfont␣10␣scalefont␣setfont" ≪ endl;
    return ps_file;
  }
  void cagd::close_postscript_file(psf &ps_file, bool with_new_page) {
    if (with_new_page ≡ true) {
      ps_file ≪ "showpage" ≪ endl;
    }
    ps_file.close();
  }
```

56, 67, 130, 133, 135, 137번 마디도 살펴보라.
이 코드는 3번 마디에서 사용된다.

5. ⟨ Declaration of **cagd** functions 5 ⟩ ≡

psf *create_postscript_file*(**string**);
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 ⟩ ≡

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include "cagd.h"
  using namespace cagd; using namespace std::chrono;
  void print_title(const char *);
  void print_title(const char *str) {
    cout << endl << endl;
    char prev = cout.fill('-');
    cout << ">> " << setw(68) << '-' << endl;
    cout << ">>" << endl;
    cout << ">>  TEST: " << str << endl;
    cout << ">>" << endl;
    cout << ">> " << setw(68) << '-' << endl;
    cout.fill(prev);
  }
  int main(int argc, char *argv[]) {
    cout <<
    "================================================================================\n" <<
    "                                                                                \n" <<
    "                     T E S T  :  C A G D    L I B R A R Y                       \n" <<
    "                                                                                \n" <<
    "================================================================================\n\n";
    ⟨ Test routines 26 ⟩;
    return 0;
  }
```

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

⟨ Definition of **point** 7 ⟩ ≡

```
struct point {
  ⟨ Data members of point 8 ⟩
  ⟨ Methods of point 11 ⟩
};
```

이 코드는 2번 마디에서 사용된다.

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

⟨ Data members of **point** 8 ⟩ ≡

```
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 ⟩ ≡

```
⟨ Properties of point 10 ⟩
⟨ Constructors and destructor of point 12 ⟩
⟨ Operators of point 14 ⟩
⟨ Other member functions of point 20 ⟩
⟨ 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 ⟩ ≡

```
size_t point::dimension() const {
  return (this→_elem).size();
}
size_t point::dim() const {
  return (this→_elem).size();
}
```

이 코드는 9번 마디에서 사용된다.

11. ⟨ Methods of **point** 11 ⟩ ≡

```
size_t dimension() const;
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 n만 주어지면, 모든 원소가 0인 n-차원 **point** 객체를 생성한다.
- **double**타입의 배열로부터 **point** 객체를 생성하는 constructor를 정의한다.

⟨ Constructors and destructor of **point** 12 ⟩ ≡

```
point::point(const point &src)
        : _elem(src._elem) { }
point::point(initializer_list<double> v)
        : _elem(vector<double>(v.begin(), v.end())) { }
point::point(const double v1, const double v2, const double v3)
        : _elem(vector<double>(3))
{
  _elem[0] = v1;
  _elem[1] = v2;
  _elem[2] = v3;
}
point::point(const size_t n)
        : _elem(vector<double>(n, 0.)) { }
point::point(const size_t n, const double *v)
        : _elem(vector<double>(n, 0.))
{
  for (size_t i = 0; i != n; i++) {
    _elem[i] = v[i];
  }
}
point::~point() { }
```

이 코드는 9번 마디에서 사용된다.

13. ⟨ Methods of **point** 11 ⟩ +≡

```
point() = delete ;
point(const point &);
point(initializer_list<double>);
point(const double, const double, const double v3 = 0.);
point(const size_t);
point(const size_t, const double *);
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 ⟩ ≡

```
void point::operator=(const point &src) {
  _elem = src._elem;
}
point &point::operator*=(const double s) {
  size_t sz = this->dim();
  for (size_t i = 0; i != sz; i++) {
    (this->_elem[i]) *= s;
  }
  return *this;
}
point &point::operator/=(const double s) {
  if (s == 0.) return *this;
  size_t sz = this->dim();
  for (size_t i = 0; i != sz; i++) {
    (this->_elem[i]) /= s;
  }
  return *this;
}
point &point::operator+=(const point &pt) {
  size_t sz_min = min(this->dim(), pt.dim());
  for (size_t i = 0; i != sz_min; i++) {
    (this->_elem[i]) += pt._elem[i];
  }
  return *this;
}
point &point::operator-=(const point &pt) {
  size_t sz_min = min(this->dim(), pt.dim());
  for (size_t i = 0; i != sz_min; i++) {
    (this->_elem[i]) -= pt._elem[i];
  }
  return *this;
}
```

18번 마디도 살펴보라.
이 코드는 9번 마디에서 사용된다.

15. ⟨ Methods of **point** 11 ⟩ +≡

```
void operator=(const point &);
point &operator*=(const double);
point &operator/=(const double);
point &operator+=(const point &);
point &operator-=(const point &);
```

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

⟨ Non-member functions for **point** 16 ⟩ ≡

```
point cagd::operator*(double s, point pt) {
  return pt *= s;
}
point cagd::operator*(point pt, double s) {
  return pt *= s;
}
point cagd::operator/(point pt, double s) {
  return pt /= s;
}
point cagd::operator+(point pt1, point pt2) {
  return pt1 += pt2;
}
point cagd::operator-(point pt1, point pt2) {
  return pt1 -= pt2;
}
point cagd::operator-(point pt1) {
  size_t sz = pt1.dim();
  cagd::point negated(sz);
  for (size_t i = 0; i != sz; i++) {
    negated._elem[i] = -pt1._elem[i];
  }
  return negated;
}
```

24번 마디도 살펴보라.
이 코드는 9번 마디에서 사용된다.

17. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

```
point operator*(double, point);
point operator*(point, double);
point operator/(point, double);
point operator+(point, point);
point operator-(point, point);
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 i가 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 ⟩ +≡

```
const double &point::operator()(const size_t &i) const {
  size_t size = _elem.size();
  if ((i < 1) || (size < i)) {
    return _elem[0];
  } else {
    return _elem[i - 1];
  }
}
double &point::operator()(const size_t &i) {
  return const_cast<double &>(static_cast<const point &>(*this)(i));
}
```

19. ⟨ Methods of **point** 11 ⟩ +≡

```
const double &operator()(const size_t &) const;
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.

⟨ Other member functions of **point** 20 ⟩ ≡

```
double point::dist(const point &pt) const {
  if (this→dim() ≠ pt.dim()) return −1.;
  size_t n = this→dim();
  double sum = 0.0;
  for (size_t i = 0; i ≠ n; i++) {
    sum += (_elem[i] − pt._elem[i]) ∗ (_elem[i] − pt._elem[i]);
  }
  return std::sqrt(sum);
}
```

22번 마디도 살펴보라.
이 코드는 9번 마디에서 사용된다.

21. ⟨ Methods of **point** 11 ⟩ +≡

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

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

⟨ Other member functions of **point** 20 ⟩ +≡

```
string point::description() const {
  stringstream buffer;
  buffer ≪ "(␣";
  for (size_t i = 0; i ≠ dim() − 1; i++) {
    buffer ≪ _elem[i] ≪ ",␣";
  }
  buffer ≪ _elem[dim() − 1] ≪ "␣)" ≪ endl;
  return buffer.str();
}
```

23. ⟨ Methods of **point** 11 ⟩ +≡

```
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 ⟩ +≡

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

25. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

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

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

⟨ Test routines 26 ⟩ ≡

```
print_title("operations␣on␣point␣type");
{
  point p0(3);
  cout << "Dimension␣of␣p0␣=␣" << p0.dim() << "␣:␣";
  for (size_t i = 0; i != p0.dim(); i++) {
    cout << p0(i + 1) << "␣␣";
  }
  cout << "\n\n";
  point p1({1., 2., 3.});
  cout << "Dimension␣of␣p1␣=␣" << p1.dim() << "␣:␣";
  for (size_t i = 0; i != p1.dim(); i++) {
    cout << p1(i + 1) << "␣␣";
  }
  cout << "\n\n";
  point p2({2., 4., 6.});
  point p3 = .5 * p1 + .5 * p2;
  cout << "p3␣=␣.5(1,2,3)␣+␣.5(2,4,6)␣=␣";
  for (size_t i = 0; i != p3.dim(); i++) {
    cout << p3(i + 1) << "␣␣";
  }
  cout << "\n\n";
  cout << "Distance␣from␣p0␣to␣p1␣=␣" << dist(p0, p1) << "\n";
  cout << "␣␣(It␣should␣be␣3.741657387)\n\n";
}
```

90, 132, 142, 159, 164, 166번 마디도 살펴보라.
이 코드는 6번 마디에서 사용된다.

27. Generic Curve. **curve** 타입은 컨트롤 포인트에 의하여 모양과 특성이 결정되는 일반적인 곡선을 의미한다. 따라서 데이타 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⟩ ≡

```
class curve {
protected:
    vector⟨point⟩ _ctrl_pts;
    mutable cagd::err_code _err;
public:
    typedef vector⟨point⟩::iterator ctrlpt_itr;
    typedef vector⟨point⟩::const_iterator const_ctrlpt_itr;
    ⟨Methods of curve 30⟩
};
```

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⟩ ≡

```
⟨Properties of curve 29⟩
⟨Access control points of curve 31⟩
⟨Constructor and destructor of curve 36⟩
⟨Methods for debugging of curve 38⟩
⟨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⟩ ≡

```
unsigned long curve::dimension( ) const {
    if (_ctrl_pts.size( ) > 0) {
        return _ctrl_pts.begin( )→dim( );
    } else {
        return 0;
    }
}
unsigned long curve::dim( ) const {
    return dimension( );
}
```

이 코드는 28번 마디에서 사용된다.

30. ⟨ Methods of **curve** 30 ⟩ ≡

```
public:
  virtual unsigned long dimension( ) const;
  virtual unsigned long dim( ) const;
  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 ⟩ ≡

```
  point curve::ctrl_pts(const size_t &i) const {
    size_t size = _ctrl_pts.size( );
    if ((i < 1) ∨ (size < i)) {
      return _ctrl_pts[0];
    } else {
      return _ctrl_pts[i];
    }
  }
  size_t curve::ctrl_pts_size( ) const {
    return _ctrl_pts.size( );
  }
```

이 코드는 28번 마디에서 사용된다.

32. ⟨ Methods of **curve** 30 ⟩ +≡

```
  point ctrl_pts(const size_t &) const;
  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 ⟩ +≡

```
virtual void write_curve_in_postscript(
        psf &,
        unsigned, float,
        int x = 1, int y = 1,
        float magnification = 1.) const = 0;
virtual void write_control_polygon_in_postscript(
        psf &,
        float,
        int x = 1, int y = 1,
        float magnification = 1.) const = 0;
virtual void write_control_points_in_postscript(
        psf &,
        float,
        int x = 1, int y = 1,
        float magnification = 1.) const = 0;
```

34. ⟨ Error codes of **cagd** 34 ⟩ ≡

```
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 ⟩ +≡

```
public:
    virtual point evaluate(const double) const = 0;
    virtual point derivative(const double) const = 0;
```

36. Constructor와 destructor에 특별한 것은 없다.

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

```
curve::curve() { }
curve::curve(const vector⟨point⟩ &pts)
        : _ctrl_pts(pts) { }
curve::curve(const list⟨point⟩ &pts)
        : _ctrl_pts(vector⟨point⟩(pts.size(), pts.begin()->dim())) {
    list⟨point⟩::const_iterator pt(pts.begin());
    for (size_t i = 0; i != pts.size(); i++) {
        _ctrl_pts[i] = *pt;
        pt++;
    }
}
curve::curve(const curve &src)
        : _ctrl_pts(src._ctrl_pts) { }
curve::~curve() { }
```

이 코드는 28번 마디에서 사용된다.

37. ⟨ Methods of **curve** 30 ⟩ +≡

```
public:
    curve();
    curve(const vector⟨point⟩ &);
    curve(const list⟨point⟩ &);
    curve(const curve &);
    virtual ~curve();
```

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

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

```
string curve::description() const {
    stringstream buffer;
    buffer << "------------------------------" << endl;
    buffer << "␣␣␣␣Description␣of␣Curve␣␣␣␣" << endl;
    buffer << "------------------------------" << endl;
    buffer << "␣␣Dimension␣of␣curve:␣" << dim() << endl;
    buffer << "␣␣Control␣points:␣" << endl;
    for (size_t i = 0; i != _ctrl_pts.size(); i++) {
        buffer << "␣␣␣␣" << _ctrl_pts[i].description();
    }
    return buffer.str();
}
```

이 코드는 28번 마디에서 사용된다.

39. ⟨ Methods of **curve** 30 ⟩ +≡

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

40. Assignment operator.

⟨ Operators of **curve** 40 ⟩ ≡

```
  curve &curve::operator=(const curve &crv) {
    _ctrl_pts = crv._ctrl_pts;
    _err = crv._err;
    return *this;
  }
```

이 코드는 28번 마디에서 사용된다.

41. ⟨ Methods of **curve** 30 ⟩ +≡

```
public:
  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 ⟩ ≡

```
class bezier : public curve {
    ⟨ Data members of bezier 43 ⟩
    ⟨ Methods of bezier 46 ⟩
};
```

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 ⟩ ≡

```
protected:
    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 ⟩ ≡

```
⟨ Properties of bezier 45 ⟩
⟨ Constructors and destructor of bezier 47 ⟩
⟨ Operators of bezier 49 ⟩
⟨ Evaluation of bezier 52 ⟩
⟨ Subdivision of bezier 58 ⟩
⟨ Degree elevation and reduction of bezier 64 ⟩
⟨ 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 ⟩ ≡

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

이 코드는 44번 마디에서 사용된다.

46. ⟨ Methods of **bezier** 46 ⟩ ≡

```
public:
  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 ⟩ ≡

```
bezier::bezier() { }

bezier::bezier(const bezier &src) {
  _degree = src._degree;
  _ctrl_pts.clear();
  for (size_t i = 0; i ≠ src._ctrl_pts.size(); ++i) {
    _ctrl_pts.push_back(src._ctrl_pts[i]);
  }
}

bezier::bezier(vector⟨point⟩ points) {
  _degree = points.size() − 1;
  _ctrl_pts.clear();
  for (size_t i = 0; i ≠ points.size(); ++i) {
    _ctrl_pts.push_back(points[i]);
  }
}

bezier::bezier(list⟨point⟩ points) {
  _degree = points.size() − 1;
  _ctrl_pts.clear();
  for (list⟨point⟩::const_iterator i = points.begin(); i ≠ points.end(); i++) {
    _ctrl_pts.push_back(*i);
  }
}

bezier::~bezier() { }
```

이 코드는 44번 마디에서 사용된다.

48. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  bezier();
  bezier(const bezier &);
  bezier(vector⟨point⟩);
  bezier(list⟨point⟩);
  virtual ~bezier();
```

49. 다른 **bezier** 객체로부터의 assignment operator.

⟨ Operators of **bezier** 49 ⟩ ≡

```
  bezier &bezier::operator=(const bezier &src) {
    _degree = src._degree;
    curve::operator=(src);
    return *this;
  }
```

이 코드는 44번 마디에서 사용된다.

50. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  bezier &operator=(const bezier &);
```

51. ⟨ Error codes of **cagd** 34 ⟩ +≡

```
  DEGREE_MISMATCH ,
```

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

⟨ Evaluation of **bezier** 52 ⟩ ≡

```
point bezier::evaluate(const double t) const {
  vector<point> coeff;
  for (size_t i = 0; i != _ctrl_pts.size(); ++i) {
    coeff.push_back(_ctrl_pts[i]);
  }
  double t1 = 1.0 - t;
  for (size_t r = 1; r != _degree + 1; r++) {
    for (size_t i = 0; i != _degree - r + 1; i++) {
      coeff[i] = t1 * coeff[i] + t * coeff[i + 1];
    }
  }
  return coeff[0];
}
point bezier::derivative(const double t) const {
  vector<point> coeff;
  for (size_t i = 0; i != _ctrl_pts.size() - 1; ++i) {
    coeff.push_back(_degree * (_ctrl_pts[i + 1] - _ctrl_pts[i]));
  }
  double t1 = 1.0 - t;
  for (size_t r = 1; r != _degree; r++) {
    for (size_t i = 0; i != _degree - r; i++) {
      coeff[i] = t1 * coeff[i] + t * coeff[i + 1];
    }
  }
  return coeff[0];
}
```

54번 마디도 살펴보라.
이 코드는 44번 마디에서 사용된다.

53. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  point evaluate(const double) const;
  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 b부터 e까지로 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 ⟩ +≡

```
double
bezier::curvature_at_zero() const {
  double dist = cagd::dist(_ctrl_pts[0], _ctrl_pts[1]);
  return 2.0 * (_degree - 1) *
  cagd::signed_area(_ctrl_pts[0], _ctrl_pts[1], _ctrl_pts[2])/(_degree * dist * dist * dist);
}
vector⟨point⟩
bezier::signed_curvature(const unsigned density, const double b, const double e) const {
    /* b: begin of the interval. e: end of the interval. */
  double delta = (e - b)/density;
  unsigned half = density/2;
  vector⟨point⟩ kappa;
  for (size_t i = 0; i ≤ density; i++) {
    double t = b + i * delta;
    bezier left(*this);
    bezier right(*this);
    if (i ≤ half) {
      subdivision(t, left, right);
      double h = right.curvature_at_zero();
      kappa.push_back(point({t, h}));
    } else {
      subdivision(t, left, right);
      double h = left.curvature_at_zero();
      kappa.push_back(point({t, std::fabs(-h)}));
    }
  }
  return kappa;
}
```

55. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  double curvature_at_zero( ) const;
  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.

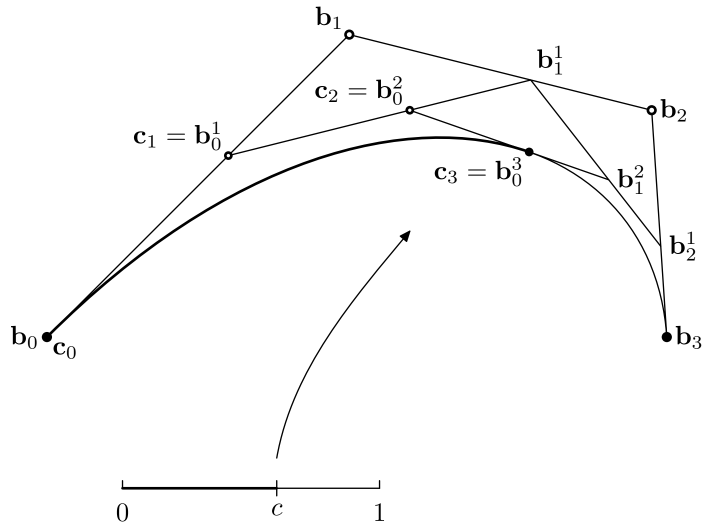
⟨ Implementation of **cagd** functions 4 ⟩ +≡

```
double cagd::signed_area(const point p1, const point p2, const point p3) {
  double area;
  area = ((p2(1) − p1(1)) ∗ (p3(2) − p1(2)) − (p2(2) − p1(2)) ∗ (p3(1) − p1(1)))/2.0;
  return area;
}
```

57. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

```
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 c인 지점에서 둘로 나누는 과정을 설명한다. Bézier 곡선을 두개로 분할하고 새로운 컨트롤 포인트들을 구하는 것은 de Casteljau 알고리즘을 적용하는 과정과 동일하다. 즉, 선분 $\mathbf{b}_i$-$\mathbf{b}_{i+1}$을 $c : 1-c$로 내분하는 점을 $\mathbf{b}_i^1(c), (i = 0, 1, 2)$이라 하고, 다시 선분 $\mathbf{b}_i^1(c)$-$\mathbf{b}_{i+1}^1(c)$를 $c : 1-c$로 내분하는 점을 $\mathbf{b}_i^2(c)(i = 0, 1)$, 또 선분 $\mathbf{b}_i^2(c)$-$\mathbf{b}_{i+1}^2(c)$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 $\mathbf{c}_0 = \mathbf{b}_0$, $\mathbf{c}_1 = \mathbf{b}_0^1(c)$, $\mathbf{c}_2 = \mathbf{b}_0^2(c)$, $\mathbf{c}_c = \mathbf{b}_0^3(c)$가 된다. $[c, 1]$ 구간도 마찬가지로 de Casteljau 알고리즘에 의해 얻어지는 중간 단계의 점들이 새로운 컨트롤 포인트가 된다.



⟨Subdivision of **bezier** 58⟩ ≡

```
void bezier::subdivision(double t, bezier &left, bezier &right) const {
  double t1 = 1.0 − t;
  vector⟨point⟩ points;    /∗ temporary store ∗/
  ⟨Obtain the right subpolygon of Bézier curve 59⟩;
  ⟨Obtain the left subpolygon of Bézier curve 61⟩;
}
```

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 ⟩ ≡

```
right._ctrl_pts.clear();
right._degree = _degree;
for (size_t i = 0; i != _ctrl_pts.size(); i++) {
  points.push_back(_ctrl_pts[i]);
}
⟨ Obtain the right subpolygon using the de Casteljau algorithm 60 ⟩;
for (size_t i = 0; i != (_degree + 1); i++) {
  right._ctrl_pts.push_back(points[i]);
}
```

이 코드는 58번 마디에서 사용된다.

60. ⟨ Obtain the right subpolygon using the de Casteljau algorithm 60 ⟩ ≡

```
for (size_t r = 1; r != _degree + 1; r++) {
  for (size_t i = 0; i != _degree - r + 1; i++) {
    points[i] = t1 * points[i] + t * points[i + 1];
  }
}
```

이 코드는 59번 마디에서 사용된다.

61. 왼쪽, 즉 파라미터 $[0, c]$ 구간에 대한 control polygon을 구한다. 방법은 오른쪽 부분 곡선을 구할때와 마찬가지인데, control point들을 temporary store에 역순으로 복사하고 t를 $1 - t$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 ⟩ ≡

```
t = 1.0 - t;
t1 = 1.0 - t1;
points.clear();
left._ctrl_pts.clear();
left._degree = _degree;
unsigned long index = _degree;
for (size_t i = 0; i != _ctrl_pts.size(); i++) {    /* Reverse order. */
  points[index--] = _ctrl_pts[i];
}
⟨ Obtain the left subpolygon using de Casteljau algorithm 62 ⟩;
for (size_t i = 0; i != _degree + 1; i++) {
  left._ctrl_pts.push_back(points[i]);
}
```

이 코드는 58번 마디에서 사용된다.

62. ⟨ Obtain the left subpolygon using de Casteljau algorithm 62 ⟩ ≡

```
for (size_t r = 1; r ≠ _degree + 1; r++) {
  for (size_t i = 0; i ≠ _degree − r + 1; i++) {
    points[i] = t1 ∗ points[i] + t ∗ points[i + 1];
  }
}
```

이 코드는 61번 마디에서 사용된다.

63. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  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 ⟩ ≡

```
void bezier::elevate_degree(unsigned long dgr) {
  if (_degree > dgr) {
    _err = DEGREE_ELEVATION_FAIL;
    return;
  }
  if (_degree ≡ dgr) {
    return;
  }
  _degree++;
  point backup_point = _ctrl_pts[0];
  unsigned long counter = 1;
  for (size_t i = 1; i ≠ _ctrl_pts.size(); ++i) {
    point tmp_point = backup_point;
    backup_point = _ctrl_pts[i];
    double ratio = double(counter)/double(_degree);
    _ctrl_pts[i] = ratio ∗ tmp_point + (1.0 − ratio) ∗ backup_point;
    counter++;
  }
  _ctrl_pts.push_back(backup_point);
  return elevate_degree(dgr);
}
```

69번 마디도 살펴보라.
이 코드는 44번 마디에서 사용된다.

65. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  void elevate_degree(unsigned long);
```

66. ⟨ Error codes of **cagd** 34 ⟩ +≡

```
  DEGREE_ELEVATION_FAIL ,
```

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

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

```
unsigned long cagd::factorial(unsigned long n) {
  if (n ≤ 0) {
    return 1UL;
  } else {
    return n * factorial(n − 1);
  }
}
```

68. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

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

```
void bezier::reduce_degree(const unsigned long dgr) {
  if (_degree < dgr) {
    _err = DEGREE_REDUCTION_FAIL;
    return;
  }
  if (_degree ≡ dgr) {
    return;
  }
  vector⟨point⟩ l2r;
  l2r.push_back(_ctrl_pts[0]);
  unsigned long counter = 1;
  for (size_t i = 1; i ≠ _ctrl_pts.size() − 1; ++i) {
    l2r.push_back((double(_degree) ∗ _ctrl_pts[i] − double(counter) ∗ (l2r.back()))/double(_degree −
        counter));
    counter++;
  }
  vector⟨point⟩ r2l_reversed;
```

```
    r2l_reversed.push_back(_ctrl_pts.back());
    counter = _degree;
    for (size_t i = _ctrl_pts.size() - 2; i ≠ 0; --i) {
      r2l_reversed.push_back((double(_degree) * (_ctrl_pts[i]) - double(_degree - counter) *
          r2l_reversed.front())/double(counter));
      counter--;
    }
    vector⟨point⟩ r2l;
    size_t r2l_reversed_size = r2l_reversed.size();
    for (size_t i = 0; i ≠ r2l_reversed_size; i++) {
      r2l.push_back(r2l_reversed.back());
      r2l_reversed.pop_back();
    }
    point backup1 = _ctrl_pts[0];
    point backup2 = _ctrl_pts.back();
    _ctrl_pts.clear();
    _ctrl_pts.push_back(backup1);
    for (size_t i = 1; i ≤ _degree - 2; ++i) {
      unsigned long combi = 0;
      for (size_t j = 0; j ≤ i; ++j) {
        combi += cagd::factorial(2 * _degree)/(cagd::factorial(2 * j) * cagd::factorial(2 * (_degree - j)));
      }
      double lambda = double(combi)/std::pow(2., 2 * _degree - 1);
      _ctrl_pts.push_back((1.0 - lambda) * l2r[i] + lambda * r2l[i]);
    }
    _ctrl_pts.push_back(backup2);
    _degree--;
    return reduce_degree(dgr);
  }
```

70. ⟨ Methods of **bezier** 46 ⟩ +≡

```
public:
  void reduce_degree(const unsigned long);
```

71. ⟨ Error codes of **cagd** 34 ⟩ +≡

```
  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 ⟩ ≡

```
void bezier::write_curve_in_postscript(
        psf &ps_file,
        unsigned step,
        float line_width,
        int x, int y,
        float magnification
        ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    ps_file << "newpath" << endl << "[] 0 setdash " << line_width << " setlinewidth" << endl;
    point pt = magnification * evaluate(0);
    ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
    for (size_t i = 1; i <= step; i++) {
        double t = double(i)/double(step);
        pt = magnification * evaluate(t);
        ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
    }
    ps_file << "stroke" << endl;
    ps_file.flags(previous_options);
}
void bezier::write_control_polygon_in_postscript(
        psf &ps_file,
        float line_width,
        int x, int y,
        float magnification
        ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    ps_file << "newpath" << endl;
    ps_file << "[] 0 setdash " << .5 * line_width << " setlinewidth" << endl;
    point pt = magnification * _ctrl_pts[0];
    ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
    for (size_t i = 1; i != _ctrl_pts.size(); ++i) {
```

```
            pt = magnification * _ctrl_pts[i];
            ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
        }
        ps_file << "stroke" << endl;
        ps_file.flags(previous_options);
    }

    void bezier::write_control_points_in_postscript(
            psf &ps_file,
            float line_width,
            int x, int y,
            float magnification
            ) const {
        ios_base::fmtflags previous_options = ps_file.flags();
        ps_file.precision(4);
        ps_file.setf(ios_base::fixed, ios_base::floatfield);
        ps_file << "0 setgray" << endl;
        ps_file << "newpath" << endl;
        point pt = magnification * _ctrl_pts[0];
        ps_file << pt(x) << "\t" << pt(y) << "\t";
        ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
        ps_file << "closepath" << endl;
        ps_file << "fill stroke" << endl;
        if (_ctrl_pts.size() > 2) {
            for (size_t i = 1; i != _ctrl_pts.size() - 1; ++i) {
                ps_file << "newpath" << endl;
                pt = magnification * _ctrl_pts[i];
                ps_file << pt(x) << "\t" << pt(y) << "\t";
                ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
                ps_file << "closepath" << endl;
                ps_file << line_width << "\t" << "setlinewidth" << endl;
                ps_file << "stroke" << endl;
            }
            ps_file << "0 setgray" << endl;
            ps_file << "newpath" << endl;
            pt = magnification * _ctrl_pts.back();
            ps_file << pt(x) << "\t" << pt(y) << "\t";
            ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
            ps_file << "closepath" << endl;
            ps_file << "fill stroke" << endl;
        }
        ps_file.flags(previous_options);
```

```
}
```

이 코드는 44번 마디에서 사용된다.

73. ⟨Methods of **bezier** 46⟩ +≡

```
void write_curve_in_postscript(
        psf &, unsigned, float, int x = 1, int y = 2,
        float magnification = 1.) const;
void write_control_polygon_in_postscript(
        psf &, float, int x = 1, int y = 2,
        float magnification = 1.) const;
void write_control_points_in_postscript(
        psf &, float, int x = 1, int y = 2,
        float magnification = 1.) const;
```

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

⟨ Definition of **piecewise_bezier_curve** 74 ⟩ ≡

```
class piecewise_bezier_curve : public curve {
protected:
  vector⟨bezier⟩ _curves;
public:
  typedef vector⟨bezier⟩::const_iterator const_curve_itr;
  typedef vector⟨bezier⟩::iterator curve_itr;
  ⟨ Methods of piecewise_bezier_curve 77 ⟩
};
```

이 코드는 2번 마디에서 사용된다.

75. **piecewise_bezier_curve** 타입의 method들은 다음과 같다.

⟨ Implementation of **piecewise_bezier_curve** 75 ⟩ ≡

```
⟨ Constructors and destructor of piecewise_bezier_curve 76 ⟩
⟨ Properties of piecewise_bezier_curve 78 ⟩
⟨ Modification of piecewise_bezier_curve 80 ⟩
⟨ Operators of piecewise_bezier_curve 82 ⟩
⟨ Degree elevation and reduction of piecewise_bezier_curve 84 ⟩
⟨ Evaluation and derivative of piecewise_bezier_curve 86 ⟩
⟨ 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 ⟩ ≡

```
piecewise_bezier_curve::piecewise_bezier_curve() { }
piecewise_bezier_curve::piecewise_bezier_curve(const piecewise_bezier_curve &r)
: curve::curve(r), _curves(r._curves) { }
piecewise_bezier_curve::~piecewise_bezier_curve() { }
```

이 코드는 75번 마디에서 사용된다.

77. ⟨ Methods of **piecewise_bezier_curve** 77 ⟩ ≡

```
public:
  piecewise_bezier_curve();
  piecewise_bezier_curve(const piecewise_bezier_curve &);
  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⟩ ≡

```
size_t piecewise_bezier_curve::count() const {
  return _curves.size();
}
unsigned long piecewise_bezier_curve::dimension() const {
  if (_curves.size() ≠ 0) {
    return _curves.begin()→dimension();
  } else {
    return 0;
  }
}
unsigned long piecewise_bezier_curve::dim() const {
  return dimension();
}
unsigned long piecewise_bezier_curve::degree() const {
  unsigned long dgr = 0;
  for (const_curve_itr crv = _curves.begin(); crv ≠ _curves.end(); crv++) {
    if (crv→degree() > dgr) {
      dgr = crv→degree();
    }
  }
  return dgr;
}
```

이 코드는 75번 마디에서 사용된다.

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

```
public:
  size_t count() const;
  unsigned long dimension() const;
  unsigned long dim() const;
  unsigned long degree() const;
```

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

⟨ Modification of **piecewise_bezier_curve** 80 ⟩ ≡

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

이 코드는 75번 마디에서 사용된다.

81. ⟨ Methods of **piecewise_bezier_curve** 77 ⟩ +≡

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

82. Operators of **piecewise_bezier_curve**.

⟨ Operators of **piecewise_bezier_curve** 82 ⟩ ≡

```
piecewise_bezier_curve &piecewise_bezier_curve::operator=(const piecewise_bezier_curve
      &crv) {
  curve::operator=(crv);
  _curves = crv._curves;
  return *this;
}
```

이 코드는 75번 마디에서 사용된다.

83. ⟨ Methods of **piecewise_bezier_curve** 77 ⟩ +≡

```
public:
  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 ⟩ ≡

```
void piecewise_bezier_curve::elevate_degree(const unsigned long dgr) {
  for (curve_itr crv = _curves.begin(); crv ≠ _curves.end(); crv++) {
    crv→elevate_degree(dgr);
  }
}

void piecewise_bezier_curve::reduce_degree(const unsigned long dgr) {
  for (curve_itr crv = _curves.begin(); crv ≠ _curves.end(); crv++) {
    crv→reduce_degree(dgr);
  }
}
```

이 코드는 75번 마디에서 사용된다.

85. ⟨Methods of **piecewise_bezier_curve** 77⟩ +≡

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

86. **piecewise_bezier_curve** 타입의 evaluation과 derivative를 구하는 method를 정의한다. 먼저 주어진 인자 u의 값을 보고 몇 번째 **bezier** 곡선에서 값을 구할지 결정한다. **piecewise_bezier_curve** 객체에 Bézier 곡선이 n개 포함되어 있다면, $0 \le u \le n$이어야 한다. 만약 객체 내에 곡선이 하나도 없거나, u가 적절한 범위 밖의 값으로 주어지면 0을 반환한다.

⟨Evaluation and derivative of **piecewise_bezier_curve** 86⟩ ≡

```
point piecewise_bezier_curve::evaluate(const double u) const {
  if (_curves.size() ≡ 0) return cagd::point(2);
  double max_u = static_cast⟨double⟩(_curves.size());
  if ((u < 0.) ∨ (max_u < u)) return cagd::point(dimension());
  size_t index;
  if (u ≡ max_u) {
    index = static_cast⟨long⟩(u) − 1;
  } else {
    index = static_cast⟨long⟩(std::floor(u));
  }
  return _curves[index].evaluate(u);
}
point piecewise_bezier_curve::derivative(const double u) const {
  if (_curves.size() ≡ 0) return cagd::point(2);
  double max_u = static_cast⟨double⟩(_curves.size());
  if ((u < 0.) ∨ (max_u < u)) return cagd::point(dimension());
  size_t index;
  if (u ≡ max_u) {
    index = static_cast⟨long⟩(u) − 1;
  } else {
    index = static_cast⟨long⟩(std::floor(u));
  }
  return _curves[index].derivative(u);
}
```

이 코드는 75번 마디에서 사용된다.

87. ⟨Methods of **piecewise_bezier_curve** 77⟩ +≡

```
public:
  point evaluate(const double) const;
  point derivative(const double) const;
```

88. **piecewise_bezier_curve** 타입의 PostScript 출력을 위한 method들이다.

⟨ PostScript output of **piecewise_bezier_curve** 88 ⟩ ≡

```
void piecewise_bezier_curve::write_curve_in_postscript(
        psf &ps_file,
        unsigned step,
        float line_width,
        int x, int y,
        float magnification
        ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    for (const_curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
        ps_file << "newpath" << endl << "[] 0 setdash " << line_width << " setlinewidth" << endl;
        point pt = magnification * (crv->evaluate(0));
        ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
        for (size_t i = 1; i <= step; i++) {
            double t = double(i)/double(step);
            pt = magnification * (crv->evaluate(t));
            ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
        }
        ps_file << "stroke" << endl;
    }
    ps_file.flags(previous_options);
}
void piecewise_bezier_curve::write_control_polygon_in_postscript(
        psf &ps_file,
        float line_width,
        int x, int y,
        float magnification
        ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    for (const_curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
        ps_file << "newpath" << endl;
        ps_file << "[] 0 setdash " << .5 * line_width << " setlinewidth" << endl;
        point pt = magnification * (crv->ctrl_pts(0));
        ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
        for (size_t i = 1; i != crv->ctrl_pts_size(); ++i) {
```

```
            pt = magnification * (crv->ctrl_pts(i));
            ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
          }
          ps_file << "stroke" << endl;
        }
        ps_file.flags(previous_options);
      }
      void piecewise_bezier_curve::write_control_points_in_postscript(
              psf &ps_file,
              float line_width,
              int x, int y,
              float magnification
              ) const {
        ios_base::fmtflags previous_options = ps_file.flags();
        ps_file.precision(4);
        ps_file.setf(ios_base::fixed, ios_base::floatfield);
        for (const_curve_itr crv = _curves.begin(); crv != _curves.end(); crv++) {
          ps_file << "0 setgray" << endl;
          ps_file << "newpath" << endl;
          point pt = magnification * (crv->ctrl_pts(0));
          ps_file << pt(x) << "\t" << pt(y) << "\t";
          ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
          ps_file << "closepath" << endl;
          ps_file << "fill stroke" << endl;
          if (crv->ctrl_pts_size() > 2) {
            for (size_t i = 1; i != crv->ctrl_pts_size() - 1; ++i) {
              ps_file << "newpath" << endl;
              pt = magnification * (crv->ctrl_pts(i));
              ps_file << pt(x) << "\t" << pt(y) << "\t";
              ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
              ps_file << "closepath" << endl;
              ps_file << line_width << "\t" << "setlinewidth" << endl;
              ps_file << "stroke" << endl;
            }
            ps_file << "0 setgray" << endl;
            ps_file << "newpath" << endl;
            pt = magnification * (crv->ctrl_pts(crv->ctrl_pts_size() - 1));
            ps_file << pt(x) << "\t" << pt(y) << "\t";
            ps_file << (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl;
            ps_file << "closepath" << endl;
            ps_file << "fill stroke" << endl;
```

```
            }
        }
        ps_file.flags(previous_options);
    }
```

이 코드는 75번 마디에서 사용된다.

89. ⟨Methods of **piecewise_bezier_curve** 77⟩ +≡

```
public:
    void write_curve_in_postscript(
            psf &, unsigned, float, int x = 1, int y = 2,
            float magnification = 1.) const;
    void write_control_polygon_in_postscript(
            psf &, float, int x = 1, int y = 2,
            float magnification = 1.) const;
    void write_control_points_in_postscript(
            psf &, float, int x = 1, int y = 2,
            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 ⟩ +≡

```
print_title("piecewise bezier curve");
{
  piecewise_bezier_curve curves;
  vector<point> ctrl_pts;
  ⟨ Build-up 3rd brush 91 ⟩;
  ⟨ Build-up 2nd, 4th, and 5th brush 92 ⟩;
  ⟨ Build-up 1st brush 93 ⟩;
  ⟨ Build-up 6th, 7th, 8th, and 9th brush (outer part) 94 ⟩;
  ⟨ Build-up 6th, 7th, 8th, and 9th brush (inner part) 95 ⟩;
  psf file = create_postscript_file("untouched.ps");   /* Draw original outline. */
  curves.write_curve_in_postscript(file, 100, 1.);
  curves.write_control_polygon_in_postscript(file, 1.);
  curves.write_control_points_in_postscript(file, 1.);
  close_postscript_file(file, true);
  unsigned long deg = curves.degree();   /* Degree elevation. */
  curves.elevate_degree(deg);
  file = create_postscript_file("degree_elevated.ps");
  curves.write_curve_in_postscript(file, 100, 1.);
  curves.write_control_polygon_in_postscript(file, 1.);
  curves.write_control_points_in_postscript(file, 1.);
  close_postscript_file(file, true);
  curves.reduce_degree(3);   /* Degree reduction. */
  file = create_postscript_file("degree_reduced.ps");
  curves.write_curve_in_postscript(file, 100, 1.);
  curves.write_control_polygon_in_postscript(file, 1.);
  curves.write_control_points_in_postscript(file, 1.);
  close_postscript_file(file, true);
}
```

91. ⟨ Build-up 3rd brush 91 ⟩ ≡

```
ctrl_pts.push_back(point({183, 416}));   /* 1st curve */
ctrl_pts.push_back(point({184, 415}));
ctrl_pts.push_back(point({185, 413}));
ctrl_pts.push_back(point({186, 412}));
ctrl_pts.push_back(point({186, 411}));
ctrl_pts.push_back(point({186, 409}));
ctrl_pts.push_back(point({184, 405}));
ctrl_pts.push_back(point({180, 401}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({180, 401}));   /* 2nd curve */
ctrl_pts.push_back(point({176, 397}));
ctrl_pts.push_back(point({172, 394}));
ctrl_pts.push_back(point({154, 359}));
ctrl_pts.push_back(point({140, 333}));
ctrl_pts.push_back(point({126, 312}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({126, 312}));   /* 3rd curve */
ctrl_pts.push_back(point({103, 278}));
ctrl_pts.push_back(point({79, 252}));
ctrl_pts.push_back(point({53, 235}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({53, 235}));   /* 4th curve */
ctrl_pts.push_back(point({46, 230}));
ctrl_pts.push_back(point({42, 228}));
ctrl_pts.push_back(point({37, 231}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({37, 231}));   /* 5th curve */
ctrl_pts.push_back(point({37, 223}));
ctrl_pts.push_back(point({39, 236}));
ctrl_pts.push_back(point({43, 243}));
ctrl_pts.push_back(point({45, 246}));
ctrl_pts.push_back(point({62, 266}));
ctrl_pts.push_back(point({76, 288}));
ctrl_pts.push_back(point({89, 313}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({89, 313}));   /* 6th curve */
```

```
ctrl_pts.push_back(point({102, 339}));
ctrl_pts.push_back(point({115, 369}));
ctrl_pts.push_back(point({127, 404}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({127, 404}));    /* 7th curve */
ctrl_pts.push_back(point({117, 400}));
ctrl_pts.push_back(point({107, 395}));
ctrl_pts.push_back(point({97, 392}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({97, 392}));    /* 8th curve */
ctrl_pts.push_back(point({86, 388}));
ctrl_pts.push_back(point({81, 386}));
ctrl_pts.push_back(point({74, 386}));
ctrl_pts.push_back(point({67, 388}));
ctrl_pts.push_back(point({57, 394}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({57, 394}));    /* 9th curve */
ctrl_pts.push_back(point({46, 399}));
ctrl_pts.push_back(point({41, 403}));
ctrl_pts.push_back(point({42, 406}));
ctrl_pts.push_back(point({43, 407}));
ctrl_pts.push_back(point({44, 407}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({44, 407}));    /* 10th curve */
ctrl_pts.push_back(point({46, 408}));
ctrl_pts.push_back(point({50, 409}));
ctrl_pts.push_back(point({68, 409}));
ctrl_pts.push_back(point({81, 410}));
ctrl_pts.push_back(point({94, 413}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({94, 413}));    /* 11th curve */
ctrl_pts.push_back(point({106, 416}));
ctrl_pts.push_back(point({115, 419}));
ctrl_pts.push_back(point({123, 425}));
ctrl_pts.push_back(point({127, 428}));
ctrl_pts.push_back(point({135, 439}));
ctrl_pts.push_back(point({139, 441}));
```

```
ctrl_pts.push_back(point({143, 441}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({143, 441}));     /* 12th curve */
ctrl_pts.push_back(point({148, 441}));
ctrl_pts.push_back(point({156, 438}));
ctrl_pts.push_back(point({169, 429}));
ctrl_pts.push_back(point({175, 423}));
ctrl_pts.push_back(point({183, 416}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
```

이 코드는 90번 마디에서 사용된다.

92. ⟨ Build-up 2nd, 4th, and 5th brush 92 ⟩ ≡

```
ctrl_pts.push_back(point({545, 226}));   /* 13th curve */
ctrl_pts.push_back(point({547, 225}));
ctrl_pts.push_back(point({550, 223}));
ctrl_pts.push_back(point({554, 217}));
ctrl_pts.push_back(point({555, 215}));
ctrl_pts.push_back(point({555, 211}));
ctrl_pts.push_back(point({547, 208}));
ctrl_pts.push_back(point({532, 206}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({532, 206}));   /* 14th curve */
ctrl_pts.push_back(point({517, 204}));
ctrl_pts.push_back(point({501, 203}));
ctrl_pts.push_back(point({482, 203}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({482, 203}));   /* 15th curve */
ctrl_pts.push_back(point({460, 203}));
ctrl_pts.push_back(point({430, 217}));
ctrl_pts.push_back(point({392, 247}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({392, 247}));   /* 16th curve */
ctrl_pts.push_back(point({329, 299}));
ctrl_pts.push_back(point({265, 366}));
ctrl_pts.push_back(point({230, 410}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({230, 410}));   /* 18th curve */
ctrl_pts.push_back(point({230, 349}));
ctrl_pts.push_back(point({230, 288}));
ctrl_pts.push_back(point({230, 227}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({230, 227}));   /* 19th curve */
ctrl_pts.push_back(point({230, 215}));
ctrl_pts.push_back(point({228, 204}));
ctrl_pts.push_back(point({224, 193}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({224, 193}));   /* 20th curve */
```

```
    ctrl_pts.push_back(point({219, 178}));
    ctrl_pts.push_back(point({211, 171}));
    ctrl_pts.push_back(point({196, 171}));
    ctrl_pts.push_back(point({190, 176}));
    ctrl_pts.push_back(point({174, 201}));
    ctrl_pts.push_back(point({169, 208}));
    ctrl_pts.push_back(point({169, 209}));
    curves.push_back(bezier(ctrl_pts));
    ctrl_pts.clear();
    ctrl_pts.push_back(point({169, 209}));    /* 21st curve */
    ctrl_pts.push_back(point({160, 217}));
    ctrl_pts.push_back(point({152, 226}));
    ctrl_pts.push_back(point({135, 243}));
    ctrl_pts.push_back(point({131, 248}));
    ctrl_pts.push_back(point({131, 250}));
    curves.push_back(bezier(ctrl_pts));
    ctrl_pts.clear();
    ctrl_pts.push_back(point({131, 250}));    /* 22nd curve */
    ctrl_pts.push_back(point({133, 252}));
    ctrl_pts.push_back(point({135, 253}));
    ctrl_pts.push_back(point({140, 253}));
    ctrl_pts.push_back(point({149, 251}));
    ctrl_pts.push_back(point({163, 246}));
    curves.push_back(bezier(ctrl_pts));
    ctrl_pts.clear();
    ctrl_pts.push_back(point({163, 246}));    /* 23rd curve */
    ctrl_pts.push_back(point({170, 243}));
    ctrl_pts.push_back(point({175, 242}));
    ctrl_pts.push_back(point({188, 242}));
    ctrl_pts.push_back(point({192, 247}));
    ctrl_pts.push_back(point({192, 258}));
    curves.push_back(bezier(ctrl_pts));
    ctrl_pts.clear();
    ctrl_pts.push_back(point({192, 258}));    /* 24th curve */
    ctrl_pts.push_back(point({192, 342}));
    ctrl_pts.push_back(point({192, 426}));
    ctrl_pts.push_back(point({192, 509}));
    curves.push_back(bezier(ctrl_pts));
    ctrl_pts.clear();
    ctrl_pts.push_back(point({192, 509}));    /* 25th curve */
    ctrl_pts.push_back(point({192, 515}));
    ctrl_pts.push_back(point({192, 519}));
```

```
ctrl_pts.push_back(point({189, 525}));
ctrl_pts.push_back(point({186, 526}));
ctrl_pts.push_back(point({175, 526}));
ctrl_pts.push_back(point({166, 523}));
ctrl_pts.push_back(point({154, 517}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({154, 517}));    /* 26th curve */
ctrl_pts.push_back(point({143, 511}));
ctrl_pts.push_back(point({134, 508}));
ctrl_pts.push_back(point({124, 508}));
ctrl_pts.push_back(point({117, 510}));
ctrl_pts.push_back(point({107, 512}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({107, 512}));    /* 27th curve */
ctrl_pts.push_back(point({98, 515}));
ctrl_pts.push_back(point({93, 518}));
ctrl_pts.push_back(point({93, 520}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({93, 520}));    /* 28th curve */
ctrl_pts.push_back(point({93, 522}));
ctrl_pts.push_back(point({95, 523}));
ctrl_pts.push_back(point({103, 526}));
ctrl_pts.push_back(point({107, 527}));
ctrl_pts.push_back(point({110, 527}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({110, 527}));    /* 29th curve */
ctrl_pts.push_back(point({122, 530}));
ctrl_pts.push_back(point({134, 534}));
ctrl_pts.push_back(point({154, 541}));
ctrl_pts.push_back(point({165, 545}));
ctrl_pts.push_back(point({180, 552}));
ctrl_pts.push_back(point({183, 555}));
ctrl_pts.push_back(point({188, 560}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({188, 560}));    /* 30th curve */
ctrl_pts.push_back(point({192, 566}));
ctrl_pts.push_back(point({196, 568}));
```

```
ctrl_pts.push_back(point({204, 568}));
ctrl_pts.push_back(point({213, 562}));
ctrl_pts.push_back(point({241, 537}));
ctrl_pts.push_back(point({248, 529}));
ctrl_pts.push_back(point({248, 524}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({248, 524}));   /* 31st curve */
ctrl_pts.push_back(point({248, 521}));
ctrl_pts.push_back(point({246, 517}));
ctrl_pts.push_back(point({238, 506}));
ctrl_pts.push_back(point({235, 502}));
ctrl_pts.push_back(point({235, 501}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({235, 501}));   /* 32nd curve */
ctrl_pts.push_back(point({231, 481}));
ctrl_pts.push_back(point({230, 457}));
ctrl_pts.push_back(point({230, 437}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({230, 437}));   /* 33rd curve */
ctrl_pts.push_back(point({232.5, 433}));
ctrl_pts.push_back(point({235, 429}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({235, 429}));   /* 34th curve */
ctrl_pts.push_back(point({256, 452}));
ctrl_pts.push_back(point({280, 486}));
ctrl_pts.push_back(point({295, 515}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({295, 515}));   /* 35th curve */
ctrl_pts.push_back(point({295, 519}));
ctrl_pts.push_back(point({296, 523}));
ctrl_pts.push_back(point({298, 530}));
ctrl_pts.push_back(point({301, 531}));
ctrl_pts.push_back(point({312, 531}));
ctrl_pts.push_back(point({321, 528}));
ctrl_pts.push_back(point({334, 520}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
```

```
ctrl_pts.push_back(point({334, 520}));    /* 36th curve */
ctrl_pts.push_back(point({347, 512}));
ctrl_pts.push_back(point({354, 505}));
ctrl_pts.push_back(point({354, 499}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({354, 499}));    /* 37th curve */
ctrl_pts.push_back(point({354, 496}));
ctrl_pts.push_back(point({351, 493}));
ctrl_pts.push_back(point({340, 487}));
ctrl_pts.push_back(point({335, 484}));
ctrl_pts.push_back(point({330, 482}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({330, 482}));    /* 38th curve */
ctrl_pts.push_back(point({304, 461}));
ctrl_pts.push_back(point({274, 437}));
ctrl_pts.push_back(point({243, 416}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({243, 416}));    /* 39th curve */
ctrl_pts.push_back(point({283, 370}));
ctrl_pts.push_back(point({342, 325}));
ctrl_pts.push_back(point({413, 283}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({413, 283}));    /* 40th curve */
ctrl_pts.push_back(point({456, 262}));
ctrl_pts.push_back(point({523, 235}));
ctrl_pts.push_back(point({545, 226}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
```

이 코드는 90번 마디에서 사용된다.

93. ⟨ Build-up 1st brush 93 ⟩ ≡

```
ctrl_pts.push_back(point({245, 638}));      /* 41st curve */
ctrl_pts.push_back(point({249, 633}));
ctrl_pts.push_back(point({251, 625}));
ctrl_pts.push_back(point({251, 614}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({251, 614}));      /* 42nd curve */
ctrl_pts.push_back(point({251, 603}));
ctrl_pts.push_back(point({247, 597}));
ctrl_pts.push_back(point({240, 597}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({240, 597}));      /* 43rd curve */
ctrl_pts.push_back(point({219, 608}));
ctrl_pts.push_back(point({164, 651}));
ctrl_pts.push_back(point({151, 666}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({151, 666}));      /* 44th curve */
ctrl_pts.push_back(point({152, 667}));
ctrl_pts.push_back(point({153, 667}));
ctrl_pts.push_back(point({155, 668}));
ctrl_pts.push_back(point({156, 668}));
ctrl_pts.push_back(point({157, 668}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({157, 668}));      /* 45th curve */
ctrl_pts.push_back(point({189, 668}));
ctrl_pts.push_back(point({224, 655}));
ctrl_pts.push_back(point({245, 638}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
```

이 코드는 90번 마디에서 사용된다.

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

```
ctrl_pts.push_back(point({535, 598}));    /* 46th curve */
ctrl_pts.push_back(point({537, 596}));
ctrl_pts.push_back(point({539, 593}));
ctrl_pts.push_back(point({539, 585}));
ctrl_pts.push_back(point({537, 581}));
ctrl_pts.push_back(point({529, 568}));
ctrl_pts.push_back(point({526, 564}));
ctrl_pts.push_back(point({526, 564}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({526, 564}));    /* 47th curve */
ctrl_pts.push_back(point({526, 507}));
ctrl_pts.push_back(point({526, 451}));
ctrl_pts.push_back(point({526, 394}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({526, 394}));    /* 48th curve */
ctrl_pts.push_back(point({527, 379}));
ctrl_pts.push_back(point({528, 364}));
ctrl_pts.push_back(point({529, 348}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({529, 348}));    /* 49th curve */
ctrl_pts.push_back(point({528, 331}));
ctrl_pts.push_back(point({521, 312}));
ctrl_pts.push_back(point({510, 307}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({510, 307}));    /* 50th curve */
ctrl_pts.push_back(point({502, 307}));
ctrl_pts.push_back(point({496, 313}));
ctrl_pts.push_back(point({489, 334}));
ctrl_pts.push_back(point({488, 344}));
ctrl_pts.push_back(point({487, 357}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({487, 357}));    /* 51st curve */
ctrl_pts.push_back(point({459, 356}));
ctrl_pts.push_back(point({418, 352}));
ctrl_pts.push_back(point({408, 347}));
curves.push_back(bezier(ctrl_pts));
```

```
ctrl_pts.clear();
ctrl_pts.push_back(point({408, 347}));    /* 52nd curve */
ctrl_pts.push_back(point({408, 335}));
ctrl_pts.push_back(point({404, 330}));
ctrl_pts.push_back(point({396, 330}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({396, 330}));    /* 53rd curve */
ctrl_pts.push_back(point({382, 336}));
ctrl_pts.push_back(point({369, 360}));
ctrl_pts.push_back(point({366, 377}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({366, 377}));    /* 54th curve */
ctrl_pts.push_back(point({367, 390}));
ctrl_pts.push_back(point({371, 421}));
ctrl_pts.push_back(point({372, 440}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({372, 440}));    /* 55th curve */
ctrl_pts.push_back(point({372, 435}));
ctrl_pts.push_back(point({372, 439}));
ctrl_pts.push_back(point({372, 554}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({372, 554}));    /* 56th curve */
ctrl_pts.push_back(point({372, 564}));
ctrl_pts.push_back(point({360, 594}));
ctrl_pts.push_back(point({355, 603}));
ctrl_pts.push_back(point({353, 617}));
ctrl_pts.push_back(point({358, 617}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({358, 617}));    /* 57th curve */
ctrl_pts.push_back(point({365, 617}));
ctrl_pts.push_back(point({372, 615}));
ctrl_pts.push_back(point({385, 607}));
ctrl_pts.push_back(point({392, 603}));
ctrl_pts.push_back(point({398, 600}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({398, 600}));    /* 58th curve */
```

```
ctrl_pts.push_back(point({417, 603}));
ctrl_pts.push_back(point({443, 609}));
ctrl_pts.push_back(point({463, 613}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({463, 613}));    /* 59th curve */
ctrl_pts.push_back(point({470, 618}));
ctrl_pts.push_back(point({480, 629}));
ctrl_pts.push_back(point({487, 632}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({487, 632}));    /* 60th curve */
ctrl_pts.push_back(point({499, 627}));
ctrl_pts.push_back(point({520, 611}));
ctrl_pts.push_back(point({535, 598}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
```

이 코드는 90번 마디에서 사용된다.

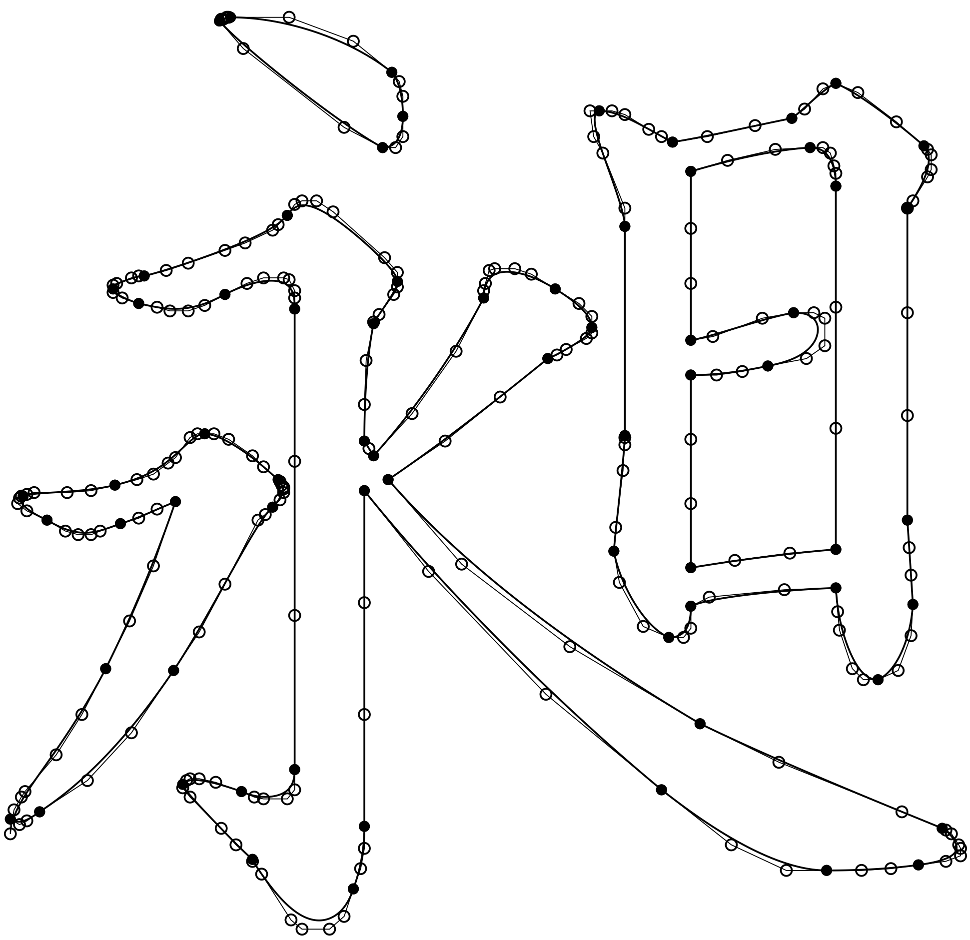
95. ⟨ Build-up 6th, 7th, 8th, and 9th brush (inner part) 95 ⟩ ≡

```
ctrl_pts.push_back(point({487, 378}));    /* 61st curve */
ctrl_pts.push_back(point({487, 444}));
ctrl_pts.push_back(point({487, 510}));
ctrl_pts.push_back(point({487, 576}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({487, 576}));    /* 62nd curve */
ctrl_pts.push_back(point({487, 583}));
ctrl_pts.push_back(point({486, 587}));
ctrl_pts.push_back(point({484, 594}));
ctrl_pts.push_back(point({480, 597}));
ctrl_pts.push_back(point({473, 597}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({473, 597}));    /* 63rd curve */
ctrl_pts.push_back(point({454, 596}));
ctrl_pts.push_back(point({428, 590}));
ctrl_pts.push_back(point({408, 584}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({408, 584}));    /* 64th curve */
ctrl_pts.push_back(point({408, 553}));
ctrl_pts.push_back(point({408, 523}));
ctrl_pts.push_back(point({408, 492}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({408, 492}));    /* 65th curve */
ctrl_pts.push_back(point({420, 494}));
ctrl_pts.push_back(point({447, 504}));
ctrl_pts.push_back(point({464, 507}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({464, 507}));    /* 66th curve */
ctrl_pts.push_back(point({475, 507}));
ctrl_pts.push_back(point({481, 504}));
ctrl_pts.push_back(point({481, 489}));
ctrl_pts.push_back(point({471, 482}));
ctrl_pts.push_back(point({450, 478}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({450, 478}));    /* 67th curve */
```

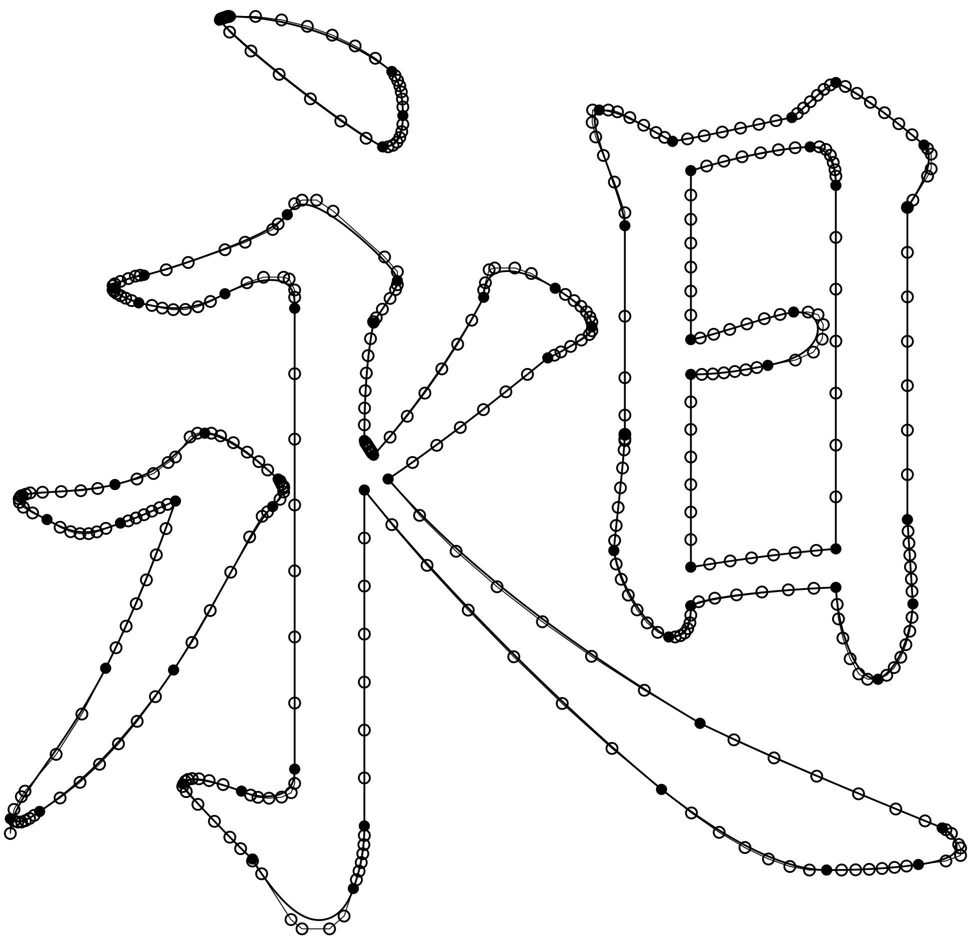
```
ctrl_pts.push_back(point({436, 475}));
ctrl_pts.push_back(point({422, 473}));
ctrl_pts.push_back(point({408, 473}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({408, 473}));     /* 68th curve */
ctrl_pts.push_back(point({408, 438}));
ctrl_pts.push_back(point({408, 403}));
ctrl_pts.push_back(point({408, 368}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
ctrl_pts.push_back(point({408, 368}));     /* 69th curve */
ctrl_pts.push_back(point({432, 372}));
ctrl_pts.push_back(point({462, 376}));
ctrl_pts.push_back(point({487, 378}));
curves.push_back(bezier(ctrl_pts));
ctrl_pts.clear();
```

이 코드는 90번 마디에서 사용된다.

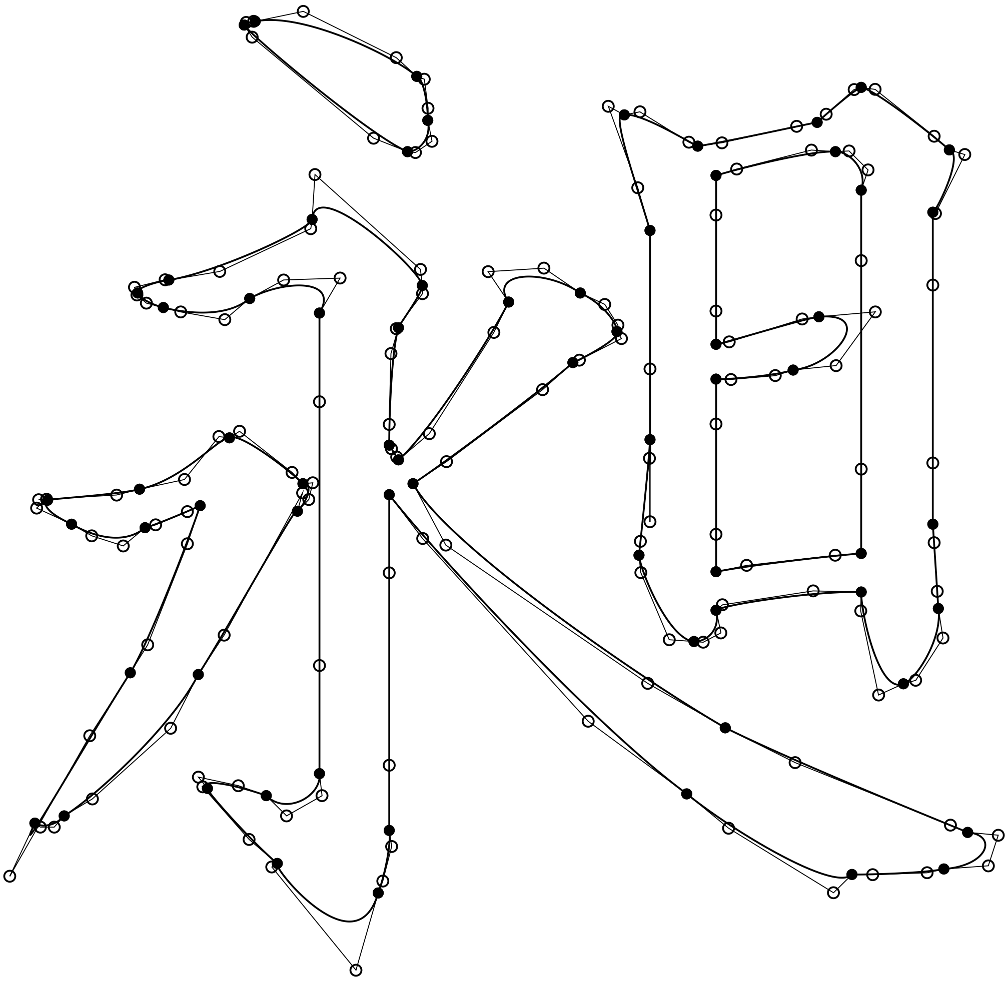
96. 예제 실행 결과. 첫 번째 그림은 traditional chinese 문자 중 하나를 골라 외곽선을 여러개의 Bézier 곡선으로 근사화한 것이다. 검은색 점은 각 곡선의 끝점을 나타내며, 흰 점은 중간의 컨트롤 포인트를, 가느다란 직선은 컨트롤 폴리곤을 나타낸다. 곡선의 차수는 3차부터 7차까지 다양하게 사용했다.



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 ⟩ ≡

```
class cubic_spline : public curve {
  ⟨ Data members of cubic_spline 100 ⟩
  ⟨ Enumerations of cubic_spline 143 ⟩
  ⟨ Methods of cubic_spline 103 ⟩
};
```

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 ⟩ ≡

```
protected:
  vector⟨double⟩ _knot_sqnc;
  mutable mpoi _mp;
  size_t _kernel_id;
protected:
  typedef vector⟨double⟩::iterator knot_itr;
  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 ⟩ ≡

```
⟨ Constructors and destructor of cubic_spline 102 ⟩
⟨ Properties of cubic_spline 104 ⟩
⟨ Operators of cubic_spline 106 ⟩
⟨ Description of cubic_spline 108 ⟩
⟨ Evaluation and derivative of cubic_spline 110 ⟩
⟨ Methods for interpolation of cubic_spline 144 ⟩
⟨ Methods to obtain a bezier curve for a segment of cubic_spline 168 ⟩
⟨ Methods to calculate curvature of cubic_spline 174 ⟩
⟨ Methods for knot insertion and removal of cubic_spline 176 ⟩
⟨ Methods for PostScript output of cubic_spline 190 ⟩
⟨ 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 ⟩ ≡

```
cubic_spline::cubic_spline(const cubic_spline &src)
        : curve(src),
        _knot_sqnc(src._knot_sqnc),
        _mp(src._mp),
        _kernel_id(src._kernel_id) { }
cubic_spline::cubic_spline(const vector⟨double⟩ &knots, const vector⟨point⟩ &pts)
        : curve(pts),
        _mp("./cspline.cl"),
        _knot_sqnc(knots),
        _kernel_id(_mp.create_kernel("evaluate_crv")) { }
cubic_spline::~cubic_spline( ) { }
```

146번 마디도 살펴보라.
이 코드는 101번 마디에서 사용된다.

103.

⟨ Methods of **cubic_spline** 103 ⟩ ≡

```
public:
cubic_spline( ) = delete ;
cubic_spline(const cubic_spline &);
cubic_spline(const vector⟨double⟩ &, const vector⟨point⟩ &);
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 ⟩ ≡

```
unsigned long cubic_spline::degree( ) const {
  return 3;
}
vector⟨double⟩ cubic_spline::knot_sequence( ) const {
  return _knot_sqnc;
}
vector⟨point⟩ cubic_spline::control_points( ) const {
  return _ctrl_pts;
}
```

이 코드는 101번 마디에서 사용된다.

105. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  unsigned long degree() const;
  vector⟨double⟩ knot_sequence() const;
  vector⟨point⟩ control_points() const;
```

106. Operators of **cubic_spline**.

⟨ Operators of **cubic_spline** 106 ⟩ ≡

```
  cubic_spline &cubic_spline::operator=(const cubic_spline &crv) {
    curve::operator=(crv);
    _knot_sqnc = crv._knot_sqnc;
    _mp = crv._mp;
    _kernel_id = crv._kernel_id;
    return *this;
  }
```

이 코드는 101번 마디에서 사용된다.

107. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  cubic_spline &operator=(const cubic_spline &);
```

108. Debugging을 위한 method를 정의한다.

⟨ Description of **cubic_spline** 108 ⟩ ≡

```
  string cubic_spline::description() const {
    stringstream buffer;
    buffer ≪ curve::description();
    buffer ≪ "␣␣Knot␣Scquence:" ≪ endl;
    for (size_t i = 0; i ≠ _knot_sqnc.size(); i++) {
      buffer ≪ "␣␣␣␣" ≪ _knot_sqnc[i] ≪ endl;
    }
    return buffer.str();
  }
```

이 코드는 101번 마디에서 사용된다.

109. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

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

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

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

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

이 때, $n+L$개의 Greville abscissas

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

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

구체적으로, degree n인 B-spline 곡선의 knot sequence u_j와 control points d_i가 있을때, $u \in [u_I, u_{I+1}) \subset [u_{n-1}, u_{L+n-1}]$를 만족하는 u에 대응하는 곡선상의 점은, $k = 1, \ldots, n-r$, $i = I-n+k+1, \ldots, I-r+1$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, r은 u가 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 u와 그것이 속하는 구간에 대한 index I를 입력으로 받는 method다. Index I는 de Boor 알고리즘을 적용하기 위하여 반드시 필요한 것이지만, 대체로 곡선상의 점을 계산하기 위하여 일일이 I까지 알아내고 그것을 함께 인자로 전달하는 것은 번거로운 일이다. 따라서 두 번째 method는 evaluation abscissa u만 인자로 전달 받으며, *find_index_in_knot_sequence*( ) 함수를 호출해서 $u[I] \le u < u[I+1]$을 만족하는 정수 I를 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 ⟩ ≡

```
point cubic_spline::evaluate(const double u, unsigned long I) const {
  const unsigned long n = 3;    /* Degree of cubic spline. */
  vector<point> tmp;
  for (size_t i = I - n + 1; i != I + 2; i++) {
    tmp.push_back(_ctrl_pts[i]);
  }
  long shifter = I - n + 1;
  for (size_t k = 1; k != n + 1; k++) {
    for (size_t i = I + 1; i != I - n + k; i--) {
      double t1 = (_knot_sqnc[i + n - k] - u)/(_knot_sqnc[i + n - k] - _knot_sqnc[i - 1]);
      double t2 = 1.0 - t1;
      tmp[i - shifter] = t1 * tmp[i - shifter - 1] + t2 * tmp[i - shifter];
    }
  }
  return tmp[I - shifter + 1];
}
point cubic_spline::evaluate(const double u) const {
  return evaluate(u, find_index_in_knot_sequence(u));
}
```

112, 118번 마디도 살펴보라.
이 코드는 101번 마디에서 사용된다.

111. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  point evaluate(const double, unsigned long) const;
  point evaluate(const double) const;
```

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

Kernel에서 계산한 m차원 공간의 N개의 점들, $\mathbf{p}_i$는 *pts*에

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

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

⟨Evaluation and derivative of **cubic_spline** 110⟩ +≡

```
vector<point>
cubic_spline::evaluate_all(const unsigned N) const {
  const unsigned n = 3;
  const unsigned L = static_cast<unsigned>(_knot_sqnc.size() - 2 * n + 1);
  const unsigned m = static_cast<unsigned>(this->dim());
  size_t pts_buffer = _mp.create_buffer(mpoi::buffer_property::READ_WRITE, N * m * sizeof(float));
  ⟨Calculate points on a cubic spline using OpenCL Kernel 114⟩;
  float *pts = new float[N * m];
  _mp.enqueue_read_buffer(pts_buffer, N * m * sizeof(float), pts);
  vector<point> crv(N, point(m));
  for (size_t i = 0; i ≠ N; i++) {
    point pt(m);
    for (size_t j = 1; j ≠ m + 1; j++) {
      pt(j) = static_cast<double>(pts[m * i + j - 1]);
    }
    crv[i] = pt;
  }
  delete[] pts;
  _mp.release_buffer(pts_buffer);
  return crv;
}
```

113. ⟨Methods of **cubic_spline** 103⟩ +≡

```
public:
  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. k개의 control point들, $\mathbf{d}_i$가 있다면 하나의 **double**형 배열에 아래와 같이 저장된다:

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

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⟩ ≡

```
const unsigned num_knots = static_cast<unsigned>(_knot_sqnc.size());
const unsigned num_ctrlpts = static_cast<unsigned>(_ctrl_pts.size());
float *knots = new float[num_knots];
float *cp = new float[num_ctrlpts * m];
size_t knots_buffer = _mp.create_buffer(mpoi::buffer_property::READ_ONLY,
    num_knots * sizeof(float));
size_t cp_buffer = _mp.create_buffer(mpoi::buffer_property::READ_ONLY,
    num_ctrlpts * m * sizeof(float));
for (size_t i = 0; i != num_knots; i++) {
  knots[i] = static_cast<float>(_knot_sqnc[i]);
}
for (size_t i = 0; i != num_ctrlpts; i++) {
  for (size_t j = 0; j != m; j++) {
    cp[i * m + j] = static_cast<float>(_ctrl_pts[i](j + 1));
  }
}
_mp.enqueue_write_buffer(knots_buffer, num_knots * sizeof(float), knots);
_mp.enqueue_write_buffer(cp_buffer, num_ctrlpts * m * sizeof(float), cp);
delete[] knots;
delete[] cp;
_mp.set_kernel_argument(_kernel_id, 0, pts_buffer);
_mp.set_kernel_argument(_kernel_id, 1, knots_buffer);
_mp.set_kernel_argument(_kernel_id, 2, cp_buffer);
```

```
_mp.set_kernel_argument(_kernel_id, 3, sizeof(unsigned), (void *) &m);
_mp.set_kernel_argument(_kernel_id, 4, sizeof(unsigned), (void *) &L);
_mp.set_kernel_argument(_kernel_id, 5, sizeof(unsigned), (void *) &N);
_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 n은 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, $[u_{n-1}, \ldots, u_{L+n-1}]$을 $N-1$개의 등간격으로 각 work item별로 자신이 계산해야 할 u 값을 계산하고, u에 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 I를 계산하는 과정에서 필요한 **if** 문을 그것과 동일한 효과를 내는 연산식으로 대체했음에 유의한다.

⟨cspline.cl 115⟩ ≡

```
#define MAX_BUFF_SIZE 30
  kernel void evaluate_crv(
          global float *crv,
          constant float *knots,
          constant float *cpts,
          unsigned d, unsigned L, unsigned N
          ) {
    private unsigned id = get_global_id(0);
    private const unsigned n = 3;
    private float tmp[MAX_BUFF_SIZE];
    private const float du = (knots[L + n − 1] − knots[n − 1])/(float)(N − 1);
    private float u = knots[n − 1] + id * du;
    private unsigned I = n − 1;
    for (private unsigned i = n; i ≠ L + n − 1; i++) {
      I += (convert_int(sign(u − knots[i])) + 1) ≫ 1;    /* If knots[i] < u, increment I. */
    }
    for (private unsigned i = 0; i ≠ n + 1; i++) {
      for (private unsigned j = 0; j ≠ d; j++) {
        tmp[i * d + j] = cpts[(i + I − n + 1) * d + j];
      }
    }
    private unsigned shifter = I − n + 1;
    for (private unsigned k = 1; k ≠ n + 1; k++) {
      for (private unsigned i = I + 1; i ≠ I − n + k; i−−) {
        private float t = (knots[i + n − k] − u)/(knots[i + n − k] − knots[i − 1]);
        for (private unsigned j = 0; j ≠ d; j++) {
          tmp[(i − shifter) * d + j] = t * tmp[(i − shifter − 1) * d + j] + (1. − t) * tmp[(i − shifter) * d + j];
        }
      }
```

```
    }
    for (private unsigned j = 0; j != d; j++) {
      crv[id * d + j] = tmp[n * d + j];
    }
  }
```

116. 어떤 scalar 값이 주어졌을 때, 그것이 knot sequence의 몇 번째 knot과 그 다음 knot 사이에 들어가는 값인지 찾아내는 method를 정의한다. 즉, u가 주어지면, $u_i \le u < u_{i+1}$을 만족하는 인덱스 i를 찾는 것이다. 만약 조건을 만족하는 i가 없으면, `SIZE_MAX`를반환한다. 이 method는 non-decreasing knot sequence를 가정하며, 만약 u가 knot sequence의 마지막 값과 같다면 조건식이 만족되지 않으므로 sequence를 뒤에서부터 거슬러 $u_i \le u \le u_{i+1}$을 만족하는 i를 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⟩ ≡

```
  size_t
  cubic_spline::find_index_in_sequence(
        const double u,
        const vector<double> sqnc
        ) const {
    if (u == sqnc.back()) {
      for (size_t i = sqnc.size() - 2; i != SIZE_MAX; i--) {
        if (sqnc[i] != u) {
          return i;
        }
      }
    }
    for (size_t i = 0; i != sqnc.size() - 1; i++) {
      if ((sqnc[i] <= u) && (u < sqnc[i + 1])) {
        return i;
      }
    }
    return SIZE_MAX;
  }
  size_t
  cubic_spline::find_index_in_knot_sequence(const double u) const {
    return find_index_in_sequence(u, this->knot_sqnc);
  }
```

122, 124, 126, 128, 182번 마디도 살펴보라.
이 코드는 101번 마디에서 사용된다.

117. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  size_t find_index_in_sequence(
           const double,
           const vector⟨double⟩
           ) const;
  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 ⟩ +≡

```
  point cubic_spline::derivative(const double u) const {
    ⟨ Check the range of knot value given 120 ⟩;
    vector⟨point⟩ splines, bezier_ctrlpt;
    vector⟨double⟩ knots;
    bezier_control_points(splines, knots);    /* Equivalent Bézier curves. */
    unsigned long index = find_index_in_sequence(u, knots);
    for (size_t i = index * 3; i ≤ (index + 1) * 3; i++) {
      bezier_ctrlpt.push_back(splines.at(i));
    }
    bezier bezier_curve = bezier(bezier_ctrlpt);
    double delta = knots[index + 1] − knots[index];
    double t = (u − knots[index])/delta;    /* Change coordinate from b-spline to Bézier. */
    point drv(bezier_curve.derivative(t));
    return drv/delta;    /* Transform the velocity into the u coordinate (b-spline). */
  }
```

119. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  point derivative(const double) const;
```

120. 만약 주어진 인자 u가 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 ⟩ ≡

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

이 코드는 118번 마디에서 사용된다.

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

```
  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 ⟩ +≡

```
unsigned long cubic_spline::find_multiplicity(const double u, const_knot_itr begin) const {
  const_knot_itr iter = find(begin, _knot_sqnc.end(), u);
  if (iter == _knot_sqnc.end()) {
    return 0;
  } else {
    return find_multiplicity(u, ++iter) + 1;
  }
}
unsigned long cubic_spline::find_multiplicity(const double u) const {
  return find_multiplicity(u, _knot_sqnc.begin());
}
```

123. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  unsigned long find_multiplicity(const double, const_knot_itr) const;
  unsigned long find_multiplicity(const double) const;
```

124. Knot sequence의 증분값, Δu_i를 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 ⟩ +≡

```
double cubic_spline::delta(const long i) const {
  if ((i < 0) || (_knot_sqnc.size() - 1) <= i) {
    return 0.;
  } else {
    return _knot_sqnc[i + 1] - _knot_sqnc[i];
  }
}
```

125. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  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 ⟩ +≡

```
void cubic_spline::insert_end_knots()
{
  vector⟨double⟩ newKnots;
  newKnots.push_back(_knot_sqnc[0]);
  newKnots.push_back(_knot_sqnc[0]);
  for (size_t i = 0; i ≠ _knot_sqnc.size(); ++i) {
    newKnots.push_back(_knot_sqnc[i]);
  }
  newKnots.push_back(_knot_sqnc.back());
  newKnots.push_back(_knot_sqnc.back());
  _knot_sqnc.clear();
  for (size_t i = 0; i ≠ newKnots.size(); ++i) {
    _knot_sqnc.push_back(newKnots[i]);
  }
}
```

127. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  void insert_end_knots();
```

128. Cubic spline 곡선의 control point들을 주어진 point들로 대치하는 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 ⟩ +≡

```
void cubic_spline::set_control_points(
        const point &head,
        const vector⟨point⟩ &intermediate,
        const point &tail
        ) {
  _ctrl_pts.clear();
  size_t n = intermediate.size();
  _ctrl_pts = vector⟨point⟩(2 + n, point(2));
  _ctrl_pts[0] = head;
  for (size_t i = 0; i ≠ n; i++) {
    _ctrl_pts[i + 1] = intermediate[i];
  }
  _ctrl_pts[n + 1] = tail;
}
```

129. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  void set_control_points(const point &, const vector<point> &, 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.

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

이 때 θ_i와 ϕ_i는 다음의 점화식으로부터 얻는다.

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

이 점화식들의 초기 조건은

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

이다.

정리하면, tridiagonal matrix의 역행렬을 구하는 과정은 다음과 같다:

1. θ_0와 θ_1을 이용하여 $\theta_2, \ldots, \theta_n$을 계산;
2. $\theta_n = 0$이면 행렬이 비가역이므로 계산 종료. 그렇지 않으면 나머지 단계로 진행;
3. ϕ_{n_1}과 ϕ_n을 이용하여 $\phi_{n-1}, \ldots, \phi_1$을 계산;
4. ϕ_i와 θ_i들을 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 −1을, 가역이면 0을 반환한다.

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

```
int cagd::invert_tridiagonal(
        const vector<double> &alpha,
        const vector<double> &beta,
        const vector<double> &gamma,
        vector<double> &inverse
        ) {
    size_t n = beta.size();
    vector<double> theta(n + 1, 0.);      /* From 0 to n. */
    theta[0] = 1.;
```

```
    theta[1] = beta[0];
    for (size_t i = 2; i != n + 1; i++) {
      theta[i] = beta[i - 1] * theta[i - 1] - gamma[i - 2] * alpha[i - 2] * theta[i - 2];
    }
    if (theta[n] == 0.) return -1;      /* The matrix is singular. */
    vector<double> phi(n + 1, 0.);      /* From 0 to n. */
    phi[n] = 1.;
    phi[n - 1] = beta[n - 1];
    for (size_t i = n - 1; i != 0; i--) {
      phi[i - 1] = beta[i - 1] * phi[i] - gamma[i - 1] * alpha[i - 1] * phi[i + 1];
    }
    for (size_t i = 0; i != n; i++) {
      for (size_t j = 0; j != n; j++) {
        double elem = 0.;
        if (i < j) {
          double prod = 1.;
          for (size_t k = i; k != j; k++) {
            prod *= gamma[k];
          }
          elem = pow(-1, i + j) * prod * theta[i] * phi[j + 1]/theta[n];
        }
        else if (i == j) {
          elem = theta[i] * phi[j + 1]/theta[n];
        }
        else {
          double prod = 1.;
          for (size_t k = j; k != i; k++) {
            prod *= alpha[k];
          }
          elem = pow(-1, i + j) * prod * theta[j] * phi[i + 1]/theta[n];
        }
        inverse[i * n + j] = elem;
      }
    }
    return 0;      /* No error. */
  }
```

131. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

```
int invert_tridiagonal(
        const vector<double> &,
        const vector<double> &,
        const vector<double> &,
        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 ⟩ +≡

```
print_title("inversion of a tridiagonal matrix");
{
  vector<double> alpha(3, 0.);
  alpha[0] = 3.; alpha[1] = 2.; alpha[2] = 1.;
  vector<double> beta(4, 0.);
  beta[0] = 1.; beta[1] = 4.; beta[2] = 3.; beta[3] = 3.;
  vector<double> gamma(3, 0.);
  gamma[0] = 4.; gamma[1] = 1.; gamma[2] = 4.;
  vector<double> inv(4 * 4, 0.);
  cagd::invert_tridiagonal(alpha, beta, gamma, inv);
  for (size_t i = 0; i != 4; i++) {
    for (size_t j = 0; j != 4; j++) {
      cout << inv[i * 4 + j] << "  ";
    }
    cout << endl;
  }
}
```

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 ⟩ +≡

```
vector<double> cagd::multiply(
        const vector<double> &mat,
        const vector<double> &vec
        ) {
  size_t n = vec.size();
  vector<double> mv(n, 0.);
  for (size_t i = 0; i != n; i++) {
    for (size_t k = 0; k != n; k++) {
      mv[i] += mat[i * n + k] * vec[k];
    }
  }
  return mv;
}
```

134. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

```
vector<double> multiply(
        const vector<double> &,
        const vector<double> &);
```

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

$$A\mathbf{x} = \mathbf{b}$$

에서 세 개의 **vector**⟨**double**⟩ 타입의 입력인자, l, d, u는 각각 $n \times n$ 행렬 A의 lower diagonal, diagonal, upper diagonal element들이다. l과 u는 $n-1$개, d는 n개의 원소를 가져야 한다. **vector**⟨**point**⟩ 타입의 인자 b와 x는 각각 방정식의 우변과 해를 의미한다. 방정식의 해가 유일하게 존재하면 함수는 0을, 그렇지 않으면 −1을 반환한다.

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

```
int cagd::solve_tridiagonal_system(
        const vector<double> &l,
        const vector<double> &d,
        const vector<double> &u,
        const vector<point> &b,
        vector<point> &x
        ) {
    size_t n = d.size();
    vector<double> Ainv(n * n, 0.);
    if (cagd::invert_tridiagonal(l, d, u, Ainv) != 0) return -1;
    for (size_t i = 1; i != b[0].dim() + 1; i++) {
        vector<double> r(n, 0.);
        for (size_t k = 0; k != n; k++) {
            r[k] = b[k](i);
        }
        vector<double> xi = cagd::multiply(Ainv, r);
        for (size_t k = 0; k != n; k++) {
            x[k](i) = xi[k];
        }
    }
    return 0;
}
```

136. ⟨Declaration of **cagd** functions 5⟩ +≡

```
int solve_tridiagonal_system(
        const vector<double> &,
        const vector<double> &,
        const vector<double> &,
        const vector<point> &,
        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

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

이 주어졌을 때,

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

으로 치환하면,

$$E\hat{\mathbf{x}} + f x_n = \hat{\mathbf{b}}$$
$$g^\top \hat{\mathbf{x}} + h x_n = b_n$$

이고, tridiagonal matrix E는 쉽게 역행렬을 구할 수 있으므로

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

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 α_i, β_i, γ_i들을 담고 있음을 가정한다.

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

```
int cagd::solve_cyclic_tridiagonal_system(
        const vector⟨double⟩ &alpha,
        const vector⟨double⟩ &beta,
        const vector⟨double⟩ &gamma,
        const vector⟨point⟩ &b,
        vector⟨point⟩ &x
        ) {
    size_t n = beta.size();
```

```
    vector⟨double⟩ Einv((n − 1) ∗ (n − 1), 0.);
    ⟨Calculate E^{−1} 138⟩;
    size_t dim = b[0].dim( );
    vector⟨vector⟨double⟩⟩ B(dim, vector⟨double⟩(n, 0.));
    for (size_t i = 0; i ≠ dim; i++) {
      for (size_t j = 0; j ≠ n; j++) {
        B[i][j] = b[j](i + 1);
      }
      ⟨Calculate x_n 139⟩;
      ⟨Calculate x̂ 140⟩;
      for (size_t j = 0; j ≠ n − 1; j++) {
        x[j](i + 1) = xhat[j];
      }
      x[n − 1](i + 1) = x_n;
    }
    return 0;
  }
```

138. ⟨Calculate E^{-1} 138⟩ ≡

```
  vector⟨double⟩ l = vector⟨double⟩(n − 2, 0.);
  vector⟨double⟩ d = vector⟨double⟩(n − 1, 0.);
  vector⟨double⟩ u = vector⟨double⟩(n − 2, 0.);
  for (size_t j = 0; j ≠ n − 2; j++) {
    l[j] = alpha[j + 1];
    d[j] = beta[j];
    u[j] = gamma[j];
  }
  d[n − 2] = beta[n − 2];
  if (invert_tridiagonal(l, d, u, Einv) ≠ 0) return −1;
```

이 코드는 137번 마디에서 사용된다.

139. g와 f의 특성으로 인하여

$$g^\top E^{-1} f = \gamma_n \left(\alpha_1 E^{-1}_{1,1} + \gamma_{n-1} E^{-1}_{1,n-1}\right) + \alpha_n \left(\alpha_1 E^{-1}_{n-1,1} + \gamma_{n-1} E^{-1}_{n-1,n-1}\right);$$
$$g^\top E^{-1} \hat{\mathbf{b}} = \gamma_n \left(E^{-1}_{1,1} b_1 + \cdots + E^{-1}_{1,n-1} b_{n-1}\right) + \alpha_n \left(E^{-1}_{n-1,1} b_1 + \cdots + E^{-1}_{n-1,n-1} b_{n-1}\right)$$

이다.

⟨ Calculate x_n 139 ⟩ ≡

```
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]);
double E1b = 0.;
double Enb = 0.;
for (size_t j = 0; j != n - 1; j++) {
  E1b += Einv[j] * B[i][j];
  Enb += Einv[(n - 2) * (n - 1) + j] * B[i][j];
}
double x_n_num = B[i][n - 1] - gamma[n - 1] * E1b - alpha[n - 1] * Enb;
double x_n = x_n_num / x_n_den;
```

이 코드는 137번 마디에서 사용된다.

140. ⟨ Calculate $\hat{\mathbf{x}}$ 140 ⟩ ≡

```
vector<double> bhat_fxn(n - 1, 0.);
for (size_t j = 0; j != n - 1; j++) {
  bhat_fxn[j] = B[i][j];
}
bhat_fxn[0] -= alpha[0] * x_n;
bhat_fxn[n - 2] -= gamma[n - 2] * x_n;
vector<double> xhat = multiply(Einv, bhat_fxn);
```

이 코드는 137번 마디에서 사용된다.

141. ⟨ Declaration of **cagd** functions 5 ⟩ +≡

```
int solve_cyclic_tridiagonal_system(
        const vector<double> &,
        const vector<double> &,
        const vector<double> &,
        const vector<point> &,
        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 ⟩ +≡

```
print_title("cyclic tridiagonal system");
{
  vector<double> alpha(7, 1.);
  vector<double> beta(7, 2.);
  vector<double> gamma(7, 1.);
  vector<point> b(7, point(2));
  b[0] = point({1., 7.});
  b[1] = point({2., 6.});
  b[2] = point({3., 5.});
  b[3] = point({4., 4.});
  b[4] = point({5., 3.});
  b[5] = point({6., 2.});
  b[6] = point({7., 1.});
  vector<point> x(7, point(2));
  solve_cyclic_tridiagonal_system(alpha, beta, gamma, b, x);
  cout << "x = " << endl;
  for (size_t i = 0; i != 7; i++) {
    cout << "[  " << x[i](1) << " ,  " << x[i](2) << "  ]" << endl;
  }
}
```

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_condition은 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 ⟩ ≡

```
public:
  enum class parametrization {
    uniform,        /* uniform parametrization */
    chord_length,   /* chord length parametrization */
    centripetal,    /* centripetal parametrization */
    function_spline /* spline function, i.e., knot sequence = x coords. */
  };
  enum class end_condition {
    clamped,    /* claped end condition */
    bessel,     /* Bessel end condition */
    quadratic,  /* quadratic end condition */
    not_a_knot, /* not-a-knot end condition */
    natural,    /* natural end condition */
    periodic    /* periodic end condition */
  };
```

이 코드는 99번 마디에서 사용된다.

144. Cubic spline 보간은 데이터 포인트, $\mathbf{p}_0, \ldots, \mathbf{p}_L$이 주어져 있을 때, 그 데이터 포인트들을 지나면서 C^2 연속성 조건을 만족하는 spline curve의 컨트롤 포인트, $\mathbf{d}_{-1}, \ldots, \mathbf{d}_{L+1}$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 $\mathbf{p}_0, \ldots, \mathbf{p}_L$ 뿐만 아니라 knot sequence, $u_0, \ldots, u_L$, 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 $3, 1, \ldots, 1, 3$을 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*( ) 메쏘드로 구현한다. 이것은 모든 종류의 보간 문제를 해결하는 engine이다.

interpolate( ) 메쏘드는 주어진 데이터 포인트의 갯수가 0이면 knot sequence와 control point를 모두 비워버린 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*( ) 메쏘드들을 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⟩ ≡

```
void
cubic_spline::_interpolate(const vector⟨point⟩ &p, parametrization scheme, end_condition
        cond, const point &initial, const point &end) {
  _knot_sqnc.clear();
  _ctrl_pts.clear();
  if (p.size() ≡ 0) {    /* No data point given. */
  }
  else if (p.size() ≡ 1) {    /* A single data point. Trivial. */
    _knot_sqnc.push_back(0.);
    _knot_sqnc.push_back(0.);
    _knot_sqnc.push_back(0.);
    _ctrl_pts.push_back(p[0]);
    _ctrl_pts.push_back(p[0]);
    _ctrl_pts.push_back(p[0]);
  }
  else {    /* More than or equal to 2 points given. */
```

```
      ⟨ Generate knot sequence according to given parametrization scheme 148 ⟩;
      ⟨ Setup equations of cubic spline interpolation 154 ⟩;
      ⟨ Modify equations according to end conditions and solve them 155 ⟩;
      insert_end_knots( );
    }
  }
```

이 코드는 101번 마디에서 사용된다.

145. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  void _interpolate(const vector⟨point⟩ &, parametrization, end_condition, const point &, const
      point &);
```

146. 한편, 데이터 포인트들이 주어졌을 때 그것들을 보간하는 cubic spline 곡선을 바로 생성하는 constructor가 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 ⟩ +≡

```
  cubic_spline::cubic_spline(const vector⟨point⟩ &p, end_condition cond, parametrization
        scheme)
        : curve(p),
        _mp("./cspline.cl"),
        _kernel_id(_mp.create_kernel("evaluate_crv"))
  {
    point one_third(2./3. * (*(p.begin( ))) + 1./3. * (p.back( )));
    point two_third(1./3. * (*(p.begin( ))) + 2./3. * (p.back( )));
    _interpolate(p, scheme, cond, one_third, two_third);
  }
  cubic_spline::cubic_spline(const vector⟨point⟩ &p, const point i, const point e, parametrization
        scheme)
        : curve(p),
        _mp("./cspline.cl"),
        _kernel_id(_mp.create_kernel("evaluate_crv"))
  {
    _interpolate(p, scheme, end_condition::clamped, i, e);
  }
```

147. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  cubic_spline(const vector⟨point⟩ &, end_condition cond = end_condition::not_a_knot,
      parametrization scheme = parametrization::chord_length);
  cubic_spline(const vector⟨point⟩ &, 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 ⟩ ≡

```
switch (scheme) {
case parametrization::uniform: {
    ⟨ Uniform parametrization of knot sequence 150 ⟩;
  }
  break;
case parametrization::chord_length: {
    ⟨ Chord length parametrization of knot sequence 151 ⟩;
  }
  break;
case parametrization::centripetal: {
    ⟨ Centripetal parametrization of knot sequence 152 ⟩;
  }
  break;
case parametrization::function_spline: {
    ⟨ Function spline parametrization of knot sequence 153 ⟩;
  }
  break;
default:
  _err = UNKNOWN_PARAMETRIZATION;
  return;
}
```

이 코드는 144번 마디에서 사용된다.

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

```
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 ⟩ ≡

```
for (size_t i = 0; i ≠ p.size(); i++) {
  _knot_sqnc.push_back(double(i));
}
```

이 코드는 148번 마디에서 사용된다.

151. Chord length parametrization: data point, x_i들 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 ⟩ ≡

```
_knot_sqnc.push_back(0.);    /* u_0 */
double sum_delta = 0.;
for (size_t i = 0; i ≠ p.size() − 1; i++) {
  double delta = cagd::dist(p[i], p[i + 1]);    /* Δ_i */
  sum_delta += delta;
  _knot_sqnc.push_back(sum_delta);
}
if (sum_delta ≠ 0.) {    /* Normalize knot sequence so that u_L = 1. */
  for (knot_itr i = _knot_sqnc.begin(); i ≠ _knot_sqnc.end(); i++) {
    *i /= sum_delta;
  }
}
```

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 $\|x_{i+1} - x_i\| = \Delta x_i$일때,

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

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 ⟩ ≡

```
double sum_delta = 0.;
_knot_sqnc.push_back(sum_delta);
for (size_t i = 0; i ≠ p.size() − 1; i++) {
  double delta = sqrt(cagd::dist(p[i], p[i + 1]));
  sum_delta += delta;
  _knot_sqnc.push_back(sum_delta);
}
if (sum_delta ≠ 0.) {  /* Normalize knot sequence so that u_L = 1. */
  for (size_t i = 0; i ≠ _knot_sqnc.size(); i++) {
    _knot_sqnc[i] /= sum_delta;
  }
}
```

이 코드는 148번 마디에서 사용된다.

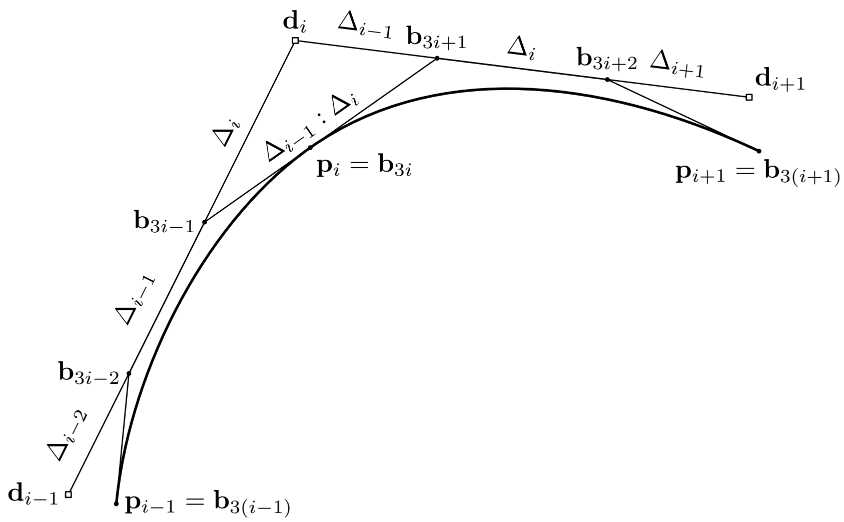
153. Function spline parametrization: 이는 data point x_i의 첫 번째 좌표들을 knot sequence로 설정하는 것이다. 주로 2차원 평면상의 점 $x_i = (u_i, v_i)$들이 있을 때, u 축에 대한 함수로써의 v를 spline interpolation할 때 사용한다.

⟨ Function spline parametrization of knot sequence 153 ⟩ ≡

```
for (size_t i = 0; i ≠ p.size(); i++) {
  _knot_sqnc.push_back(p[i](1));
}
```

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,

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

이고, inner Bézier control point, $\mathbf{b}_{3i\pm1}$과 $\mathbf{p}_i$ 사이의 관계는 곡선의 C^1 연속성 조건에 의하여

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

이다. 이때, $\Delta_i = \Delta u_i$를 간략하게 쓴 것이다. 이제 C^2 연속성 조건에 의하여 spline의 컨트롤 포인트 $\mathbf{d}_i$와 $\mathbf{b}_{3i\pm1}$ 사이의 관계는

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

이다. 곡선의 양 끝부분에서는 조금 상황이 다르며

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

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⟩ ≡

```
vector⟨double⟩ a;    /* α, lower diagonal. */
vector⟨double⟩ b;    /* β, diagonal. */
vector⟨double⟩ c;    /* γ, upper diagonal. */
vector⟨point⟩ r;    /* r, right hand side. */
unsigned long L = p.size() − 1;
b.push_back(1.0);    /* First row. */
c.push_back(0.0);
r.push_back(initial);
for (size_t i = 1; i ≠ L; i++) {
  double delta_im2 = delta(i − 2);
  double delta_im1 = delta(i − 1);
  double delta_i = delta(i);
  double delta_ip1 = delta(i + 1);
  double alpha_i = delta_i ∗ delta_i/(delta_im2 + delta_im1 + delta_i);
  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);
  double gamma_i = delta_im1 ∗ delta_im1/(delta_im1 + delta_i + delta_ip1);
  a.push_back(alpha_i);
  b.push_back(beta_i);
  c.push_back(gamma_i);
  r.push_back((delta_im1 + delta_i) ∗ p[i]);
```

```
    }
    a.push_back(0.);
    b.push_back(1.);
    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 ⟩ ≡

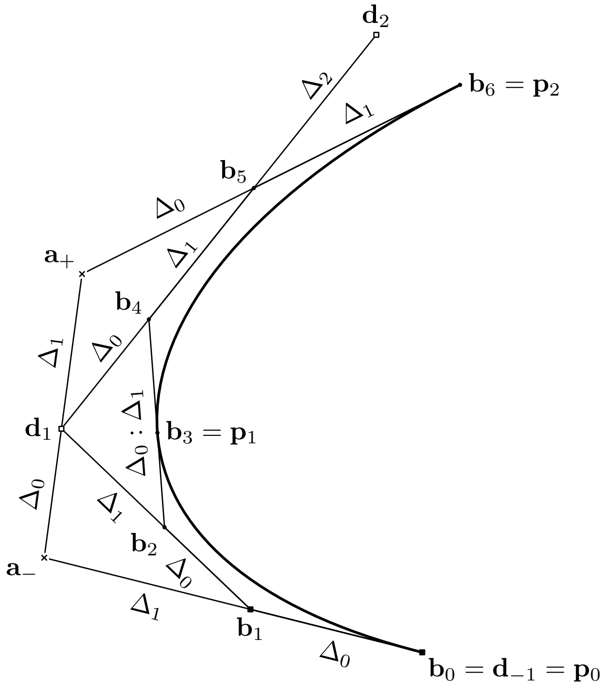
```
switch (cond) {
case end_condition::not_a_knot: {
    ⟨ Modify equations according to not-a-knot end condition 157 ⟩;
  }
case end_condition::clamped: {     /* No modification required. */
    vector⟨point⟩ x(L + 1, point(p[0].dim()));
    if (solve_tridiagonal_system(a, b, c, r, x) ≠ 0) {
      _err = TRIDIAGONAL_NOT_SOLVABLE;
      return;
    }
    set_control_points(p[0], x, p[L]);
  }
  break;
case end_condition::periodic: {
    ⟨ Modify equations according to periodic end condition 158 ⟩;
    vector⟨point⟩ x(L, point(p[0].dim()));
    if (solve_cyclic_tridiagonal_system(a, b, c, r, x) ≠ 0) {
      _err = TRIDIAGONAL_NOT_SOLVABLE;
      return;
    }
    point d_plus(((delta(0) + delta(1)) ∗ x[0] + delta(L − 1) ∗ x[1])/(delta(L − 1) + delta(0) + delta(1)));
    point d_minus(((delta(L − 2) + delta(L − 1)) ∗ x[0] + delta(0) ∗ x[L − 1])/(delta(L − 2) + delta(L −
        1) + delta(0)));
    vector⟨point⟩ d(L + 1, point(p[0].dim()));
    d[0] = d_plus;
    for (size_t i = 1; i ≠ L; i++) {
      d[i] = x[i];
    }
    d[L] = d_minus;
    set_control_points(p[0], d, p[L]);
  }
  break;
default:
  _err = UNKNOWN_END_CONDITION;
  return;
}
```

이 코드는 144번 마디에서 사용된다.

156. ⟨Error codes of **cagd** 34⟩ +≡

```
TRIDIAGONAL_NOT_SOLVABLE, UNKNOWN_END_CONDITION ,
```

157. Not-a-knot end condition은 곡선 양 끝에 놓인 각각 2개의 곡선 조각들이 하나의 Bézier 곡선이 되도록하는 조건이다. 보간해야 하는 데이터 포인트, $\mathbf{p}_0, \ldots, \mathbf{p}_L$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

$$\mathbf{a}_- = \frac{1}{s}\mathbf{d}_0 - \frac{1-s}{s}\mathbf{p}_0$$

$$\mathbf{a}_+ = \frac{1}{1-s}\{(1-r)\mathbf{d}_1 + r\mathbf{d}_2\} - \frac{s}{1-s}\mathbf{p}_2$$

을 세 번째 식에 대입하고 정리하면

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

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 q를 소거하면

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

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,

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

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 α_i, β_i, γ_i와 r_i를 그대로 채워 넣는다. Not-a-knot end condition은 최소 3개 이상($2 \le L$)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.

⟨Modify equations according to not-a-knot end condition 157⟩ ≡

```
if (L ≥ 2) {
```

```
    double s_i = delta(0)/(delta(0) + delta(1));
    double r_i = (delta(0) + delta(1))/(delta(0) + delta(1) + delta(2));
    double s_f = delta(L - 1)/(delta(L - 2) + delta(L - 1));
    double r_f = (delta(L - 2) + delta(L - 1))/(delta(L - 3) + delta(L - 2) + delta(L - 1));
    b[0] = 0.;    /* First row. */
    c[0] = -3 * s_i * s_i + 3 * s_i;
    r[0] = -(1 - s_i) * (1 - s_i) * (1 - s_i) * p[0] + p[1] - s_i * s_i * s_i * p[2];
    a[0] = (1 - s_i)/s_i;    /* Second row. */
    b[1] = s_i/(1 - s_i) * (1 - r_i) - 1;
    c[1] = s_i * r_i/(1 - s_i);
    r[1] = (1 - s_i) * (1 - s_i)/s_i * p[0] + s_i * s_i/(1 - s_i) * p[2];
    a[L - 2] = s_f * r_f/(1 - s_f);    /* Second to the last row. */
    b[L - 1] = s_f/(1 - s_f) * (1 - r_f) - 1;
    c[L - 1] = (1 - s_f)/s_f;
    r[L - 1] = s_f * s_f/(1 - s_f) * p[L - 2] + (1 - s_f) * (1 - s_f)/s_f * p[L];
    a[L - 1] = -3 * s_f * s_f + 3 * s_f;    /* Last row. */
    b[L] = 0.;
    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];
}
```

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

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

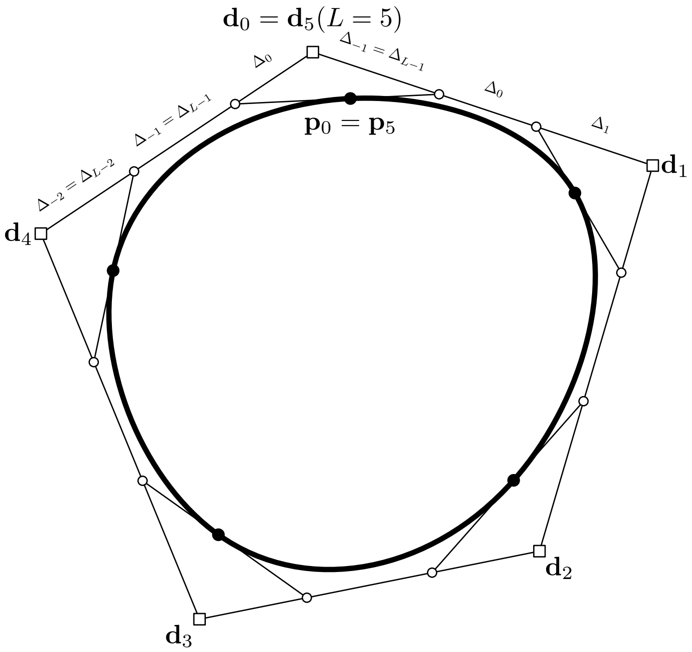
의 해를 구함으로써 얻을 수 있다. 이 때

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

이고,

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

이다. α_i, β_i, γ_i, $\mathbf{r}_i$의 정의는 앞의 C^2 조건으로부터 유도되는 식을 따르는데, 새로운 Δ_i의 정의로 인하여 α_0, α_1, β_0, β_1, β_{L-1}, γ_0, γ_{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 ⟩ ≡

```
for (size_t i = L − 1; i ≠ 0; i−−) {    /* Modify α. */
  a[i] = a[i − 1];
}
a[0] = delta(0) * delta(0)/(delta(L − 2) + delta(L − 1) + delta(0));
a[1] = delta(1) * delta(1)/(delta(L − 1) + delta(0) + delta(1));
b.pop_back();    /* Modify β. */
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));
```

$b[1] = \mathit{delta}(1) * (\mathit{delta}(L-1) + \mathit{delta}(0))/(\mathit{delta}(L-1) + \mathit{delta}(0) + \mathit{delta}(1)) + \mathit{delta}(0) * (\mathit{delta}(1) + \mathit{delta}(2))/(\mathit{delta}(0) + \mathit{delta}(1) + \mathit{delta}(2));$

$b[L-1] = \mathit{delta}(L-1) * (\mathit{delta}(L-3) + \mathit{delta}(L-2))/(\mathit{delta}(L-3) + \mathit{delta}(L-2) + \mathit{delta}(L-1)) + \mathit{delta}(L-2) * (\mathit{delta}(L-1) + \mathit{delta}(0))/(\mathit{delta}(L-2) + \mathit{delta}(L-1) + \mathit{delta}(0));$

$c[0] = \mathit{delta}(L-1) * \mathit{delta}(L-1)/(\mathit{delta}(L-1) + \mathit{delta}(0) + \mathit{delta}(1));$ /* Modify γ. */

$c[L-1] = \mathit{delta}(L-2) * \mathit{delta}(L-2)/(\mathit{delta}(L-2) + \mathit{delta}(L-1) + \mathit{delta}(0));$

$r.\mathit{pop\_back}();$ /* Modify $\mathbf{r}$. */

$r[0] = (\mathit{delta}(L-1) + \mathit{delta}(0)) * p[0];$

이 코드는 155번 마디에서 사용된다.

159. Test: Cubic Spline Interpolation.

$x = \pi, \pi+1, \ldots, \pi+10$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 ⟩ +≡

```
print_title("cubic spline interpolation");
{
  ⟨ Generate example data points 160 ⟩;
  cubic_spline crv(p, cubic_spline::end_condition::not_a_knot,
      cubic_spline::parametrization::function_spline);
  psf file = create_postscript_file("sine_curve.ps");
  crv.write_curve_in_postscript(file, 100, 1., 1, 2, 40.);
  crv.write_control_polygon_in_postscript(file, 1., 1, 2, 40.);
  crv.write_control_points_in_postscript(file, 1., 1, 2, 40.);
  close_postscript_file(file, true);
  ⟨ Compare the result of interpolation with MATLAB 161 ⟩;
  cout << crv.description();
  const unsigned steps = 1000;
  vector<double> knots = crv.knot_sequence();
  double du = (knots[knots.size() - 3] - knots[2])/double(steps - 1);
  double us[steps];
  vector<point> crv_pts_s(steps, point(2));
  for (size_t i = 0; i != steps; i++) {
    us[i] = knots[2] + i * du;
  }
  auto t0 = high_resolution_clock::now();
  for (size_t i = 0; i != steps; i++) {
    crv_pts_s[i] = crv.evaluate(us[i]);
  }
  auto t1 = high_resolution_clock::now();
  cout << "Serial computation : " << duration_cast<milliseconds>(t1 - t0).count() << " msec\n";
  t0 = high_resolution_clock::now();
  vector<point> crv_pts_p = crv.evaluate_all(steps);
  t1 = high_resolution_clock::now();
  cout << "Parallel computation : " << duration_cast<milliseconds>(t1 - t0).count() <<
      " msec\n";
  double diff = 0.;
  for (size_t i = 0; i != steps; i++) {
```

```
      diff += dist(crv_pts_s[i], crv_pts_p[i]);
    }
    cout << "Mean␣difference␣between␣serial␣and␣parallel␣computation␣=␣" <<
        diff / double(steps) << endl;
  }
```

160. ⟨ Generate example data points 160 ⟩ ≡

```
  vector⟨point⟩ p;
  for (unsigned i = 0; i != 11; i++) {
    point datum = point({0, 0});
    datum(1) = static_cast⟨double⟩(i) + M_PI;
    datum(2) = sin(datum(1)) + 3.;
    p.push_back(datum);
  }
```

이 코드는 159번 마디에서 사용된다.

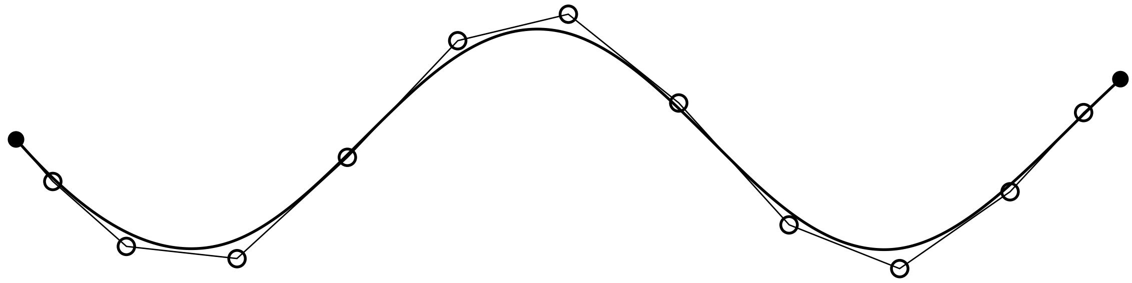
161. 보간 결과의 정확성을 검증하기 위하여 MATLAB에서 *spline*( ) 함수를 이용하여 보간한 것과 비교한다. $x = \pi, \pi + .25, \pi + .5, \ldots, \pi + 10$에 대하여 cubic spline으로 보간한 함수 $f()$로부터 $y = f(x)$를 계산한 것을 root-mean-square로 상호 비교했다.

⟨ Compare the result of interpolation with MATLAB 161 ⟩ ≡

```
  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};
  double interpolated[41];
  double u = M_PI;
  double err = 0.;
  for (size_t i = 0; i != 41; i++) {
    double y = crv.evaluate(u)(2);
    interpolated[i] = y;
    u += 0.25;
    err += (interpolated[i] - matlab_bench[i]) * (interpolated[i] - matlab_bench[i]);
  }
  err /= 41;
  err = sqrt(err);
  cout << "RMS␣error␣of␣interpolation␣(compared␣with␣MATLAB)␣=␣" << err << endl;
```

이 코드는 159번 마디에서 사용된다.

162. 실행 결과.

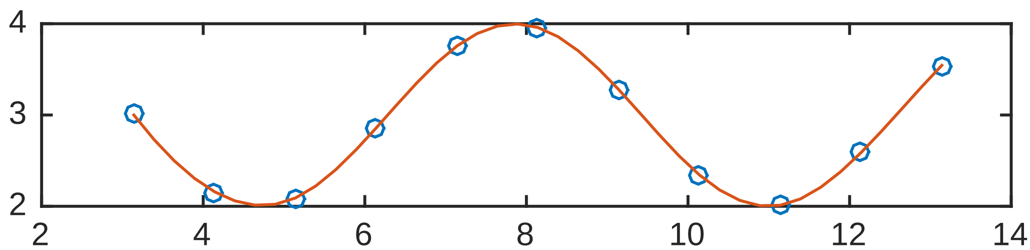


MATLAB에서 *spline*( ) 함수를 이용하여 같은 데이터를 cubic spline 보간한 것과 결과를 비교하면 다음과 같은 결과가 출력된다. 오차가 10^{-5} 오더로 발생한 것은 MATLAB에서 계산한 결과를 소수점 4째 자리까지 반올림한 것으로 가져왔기 때문이다.

```
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')
```



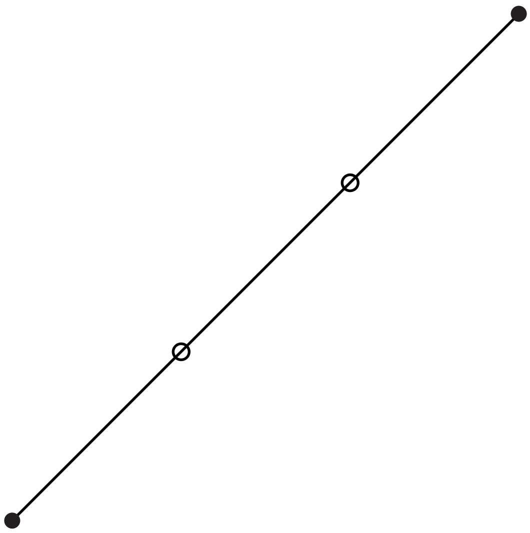
164. Test: Cubic Spline Interpolation (Degenerate Case).

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 ⟩ +≡

```
print_title("cubic␣spline␣interpolation:␣degenerate␣case");
{
  vector⟨point⟩ p;
  p.push_back(point({10, 10}));
  p.push_back(point({200, 200}));
  cubic_spline crv(p);
  psf file = create_postscript_file("line.ps");
  crv.write_curve_in_postscript(file, 30, 1., 1, 2, 1.);
  crv.write_control_polygon_in_postscript(file, 1., 1, 2, 1.);
  crv.write_control_points_in_postscript(file, 1., 1, 2, 1.);
  close_postscript_file(file, true);
}
```

165. 실행 결과.



166. Test: Periodic Cubic Spline Interpolation.

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

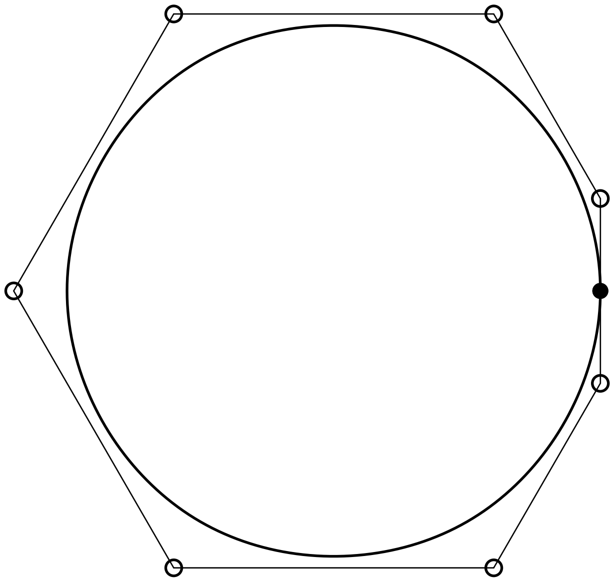
을 periodic cubic spline으로 보간한다.

⟨ Test routines 26 ⟩ +≡

```
print_title("periodic␣spline␣interpolation"); { vector<point> p;
double r = 100.;
cout << "Data␣points:" << endl;
for (size_t i = 0; i != 7; i++) {
  p.push_back(point({r * cos(2 * M_PI/6 * i) + 200., r * sin(2 * M_PI/6 * i) + 200.}));
  cout << "␣(␣" << r * cos(2 * M_PI/6 * i) + 200. << "␣,␣" << r * sin(2 * M_PI/6 * i) + 200. << "␣)" << endl;
}
cubic_spline crv(p, cubic_spline::end_condition::periodic,
    cubic_spline::parametrization::centripetal);
psf file = create_postscript_file("periodic.ps");
crv.write_curve_in_postscript(file, 200, 1., 1, 2, 1.);
crv.write_control_polygon_in_postscript(file, 1., 1, 2, 1.);
crv.write_control_points_in_postscript(file, 1., 1, 2, 1.);
close_postscript_file(file, true);
cout << crv.description();
const unsigned steps = 1000;
vector<double> knots = crv.knot_sequence();
double du = (knots[knots.size() - 3] - knots[2])/double(steps - 1);
double us[steps];
vector<point> crv_pts_s(steps, point(2));
for (size_t i = 0; i != steps; i++) {
  us[i] = knots[2] + i * du;
}
auto t0 = high_resolution_clock::now();
for (size_t i = 0; i != steps; i++) {
  crv_pts_s[i] = crv.evaluate(us[i]);
}
auto t1 = high_resolution_clock::now();
cout << "Serial␣computation␣:␣" << duration_cast<milliseconds>(t1 - t0).count() << "␣msec\n";
t0 = high_resolution_clock::now();
vector<point> crv_pts_p = crv.evaluate_all(steps);
t1 = high_resolution_clock::now();
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 나뉘는 각 조각별로 형상이 같은 Bézier 곡선으로 변환할 수 있다.

⟨Methods to obtain a **bezier** curve for a segment of **cubic_spline** 168⟩ ≡

```
void cubic_spline::bezier_control_points(vector<point> &bezier_ctrl_points, vector<double> &knot)
      const {
   bezier_ctrl_points.clear();
   knot.clear();
   ⟨Create a new knot sequence of which each knot has multiplicity of 1 170⟩;
   ⟨Check whether the curve can be broken into Bézier curves 171⟩;
   ⟨Calculate Bézier control points 172⟩;
}
```

이 코드는 101번 마디에서 사용된다.

169. ⟨Error codes of **cagd** 34⟩ +≡

```
UNABLE_TO_BREAK_INTO_BEZIER,
```

170. 모든 knot들의 multiplicity가 1이 되도록 한다. Knot sequence를 따라가며 순증가하는 knot들만 추려 낸다.

⟨ Create a new knot sequence of which each knot has multiplicity of 1 170 ⟩ ≡

```
knot.push_back(_knot_sqnc[0]);
for (size_t i = 1; i ≠ _knot_sqnc.size(); i++) {
  if (_knot_sqnc[i] > knot.back()) {
    knot.push_back(_knot_sqnc[i]);
  }
}
```

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 ⟩ ≡

```
if (knot.size() + 2 ≠ _ctrl_pts.size()) {
  _err = UNABLE_TO_BREAK_INTO_BEZIER;
  return;
}
```

이 코드는 168번 마디에서 사용된다.

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

⟨ Calculate Bézier control points 172 ⟩ ≡

```
for (size_t i = 0; i ≤ 3 * (knot.size() − 1); i++) {
  bezier_ctrl_points.push_back(point({0.0, 0.0}));
}
bezier_ctrl_points[0] = _ctrl_pts[0];    /* Special treatment on the first segment. */
bezier_ctrl_points[1] = _ctrl_pts[1];
double delta = knot[2] − knot[0];
bezier_ctrl_points[2] = ((knot[2] − knot[1]) * _ctrl_pts[1] + (knot[1] − knot[0]) * _ctrl_pts[2])/delta;
for (size_t i = 2; i ≤ knot.size() − 2; i++) {    /* Intermediate segments. */
  delta = knot[i + 1] − knot[i − 2];
  bezier_ctrl_points[3*i−1] = ((knot[i+1]−knot[i])*_ctrl_pts[i]+(knot[i]−knot[i−2])*_ctrl_pts[i+1])/delta;
  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;
}
unsigned long L = knot.size() − 1;    /* Special treatment on the last segment. */
delta = knot[L] − knot[L − 2];
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;
bezier_ctrl_points[3 * L − 1] = _ctrl_pts[L + 1];
bezier_ctrl_points[3 * L] = _ctrl_pts[L + 2];
for (size_t i = 1; i ≤ (knot.size() − 2); i++) {    /* Finally, calculate b_{3i}s. */
  delta = knot[i + 1] − knot[i − 1];
  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;
}
```

이 코드는 168번 마디에서 사용된다.

173. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  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 ⟩ ≡

```
vector⟨point⟩ cubic_spline::signed_curvature(int density) const {
  vector⟨point⟩ bezier_ctrl_points;
  vector⟨double⟩ knot;
  vector⟨point⟩ curvature;
  bezier_control_points(bezier_ctrl_points, knot);    /* Get equivalent Bézier curves. */
  for (size_t i = 0; i ≠ (knot.size() − 2); i++) {
    list⟨point⟩ cpts;    /* Control points for a section of Bézier curve. */
    cpts.clear();
    cpts.push_back(bezier_ctrl_points[3 * i]);
    cpts.push_back(bezier_ctrl_points[3 * i + 1]);
    cpts.push_back(bezier_ctrl_points[3 * i + 2]);
    cpts.push_back(bezier_ctrl_points[3 * i + 3]);
    bezier segment(cpts);
    vector⟨point⟩ kappa = segment.signed_curvature(density);
    for (size_t j = 0; j ≠ kappa.size(); j++) {
      curvature.push_back(kappa[j]);
    }
  }
  return curvature;
}
```

이 코드는 101번 마디에서 사용된다.

175. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
protected:
  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 ⟩ ≡

```
void cubic_spline::insert_knot(const double u) {
  const int n = 3;    /* Degree of cubic spline. */
  size_t index = find_index_in_knot_sequence(u);
  if (index ≡ SIZE_MAX) {
    _err = OUT_OF_KNOT_RANGE;
  }
  if ((index < n − 1) ∨ (int(_knot_sqnc.size()) − n < index)) {
    _err = NOT_INSERTABLE_KNOT;
  }
  vector⟨point⟩ new_ctrl_pts;    /* construct a new control points */
  ⟨ Copy control points for i = 0, ..., I − d + 1 178 ⟩;
  ⟨ Construct new control points by piecewise linear interpolation 179 ⟩;
  ⟨ Copy remaining control points to new control points 180 ⟩;
  _knot_sqnc.insert(_knot_sqnc.begin() + index + 1, u);
  _ctrl_pts.clear();
  _ctrl_pts = new_ctrl_pts;
}
```

184번 마디도 살펴보라.

이 코드는 101번 마디에서 사용된다.

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

```
NOT_INSERTABLE_KNOT ,
```

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

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

이 코드는 176번 마디에서 사용된다.

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

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

이 코드는 176번 마디에서 사용된다.

180. ⟨ Copy remaining control points to new control points 180 ⟩ ≡

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

이 코드는 176번 마디에서 사용된다.

181. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  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 ⟩ +≡

```
double cubic_spline::get_blending_ratio(const vector<double> &IGESKnot, long v, long r, long i)
  {
    long beta = 1;      /* set beta and determine m1 and m2 */
    long m1 = beta - r + 6 - v;
    if (m1 < 0) {
      m1 = 0;
    }
    long m2 = r - _ctrl_pts.size() + 2 + beta;
    if (m2 < 0) {
      m2 = 0;
    }
    if ((v - 1 <= i) && (i <= v - 2 + m1)) {      /* special cases to return 0 or 1 */
      return 0.;
    }
    if ((4 - m2 <= i) && (i <= 3)) {
      return 1.;
    }
    double gamma = 0.;      /* otherwise go through a laborious chore */
    for (size_t j = v - 1 + m1; j <= 4 - m2; j++) {
      double brk = bracket(IGESKnot, j + 1, 3, r);
      gamma += brk * brk;
    }
    double result = 0.;
    for (size_t j = v - 1 + m1; j <= i; j++) {
      double brk = bracket(IGESKnot, j + 1, 3, r);
      result += brk * brk;
    }
    return result/gamma;
  }
double cubic_spline::bracket(const vector<double> &IGESKnot, long a, long b, long r) {
    if (a == b + 1) {
      return 1./find_l(IGESKnot, a - 1, r);
    }
    if (a == b + 2) {
      return 1./(1. - find_l(IGESKnot, a - 1, r));
    }
```

```
    double result = 1./find_l(IGESKnot, a - 1, r);
    for (size_t i = a; i ≤ b; i++) {
      double tmp = find_l(IGESKnot, i, r);
      result *= (1. - tmp)/tmp;
    }
    return result;
  }
  double cubic_spline::find_l(const vector⟨double⟩ &IGESKnot, long j, long r) {
    return (IGESKnot[r] - IGESKnot[r - 4 + j])/(IGESKnot[r + j] - IGESKnot[r - 4 + j]);
  }
```

183. ⟨Methods of **cubic_spline** 103⟩ +≡

```
protected:
  double get_blending_ratio(const vector⟨double⟩ &, long, long, long);
  double bracket(const vector⟨double⟩ &, long, long, long);
  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⟩ +≡

```
  void cubic_spline::remove_knot(const double u) {
    vector⟨double⟩ IGESKnot;
    vector⟨point⟩ forward;
    vector⟨point⟩ backward;
    const int k = 4;
    ⟨Set multiplicity of end knots to order of this curve instead of degree 185⟩;
    size_t r = find_index_in_knot_sequence(u) + 1;
    unsigned long v = find_multiplicity(u);
    ⟨Determine forward control points 186⟩;
    ⟨Determine backward control points 187⟩;
    ⟨Blend forward and backward control points 188⟩;
    for (size_t i = r; i ≤ _knot_sqnc.size() - 1; i++) {
      _knot_sqnc[i - 1] = _knot_sqnc[i];
    }
    _knot_sqnc.pop_back();
  }
```

185. ⟨Set multiplicity of end knots to order of this curve instead of degree 185⟩ ≡

```
IGESKnot.push_back(_knot_sqnc[0]);
for (size_t i = 0; i ≠ _knot_sqnc.size(); ++i) {
  IGESKnot.push_back(_knot_sqnc[i]);
}
IGESKnot.push_back(_knot_sqnc.back());
```

이 코드는 184번 마디에서 사용된다.

186. ⟨Determine forward control points 186⟩ ≡

```
for (size_t i = 0; i ≤ r − k + v − 1; i++) {
  forward.push_back(_ctrl_pts[i]);
}
for (size_t i = r − k + v; i ≤ r − 1; i++) {
  double l = (IGESKnot[r] − IGESKnot[i])/(IGESKnot[k + i] − IGESKnot[i]);
  forward.push_back(1.0/l * _ctrl_pts[i] + (1.0 − 1.0/l) * forward[i − 1]);
}
for (size_t i = r; i ≤ _ctrl_pts.size() − 2; i++) {
  forward.push_back(_ctrl_pts[i + 1]);
}
```

이 코드는 184번 마디에서 사용된다.

187. ⟨Determine backward control points 187⟩ ≡

```
for (size_t i = 0; i ≤ _ctrl_pts.size() − 2; i++) {
  backward.push_back(cagd::point(2));
}
for (long i = _ctrl_pts.size() − 2; i ≥ r − 1; i−−) {
  backward[i] = _ctrl_pts[i + 1];
}
for (long i = r − 2; i ≥ r − k + v − 1; i−−) {
  double l = (IGESKnot[r] − IGESKnot[i + 1])/(IGESKnot[k + i + 1] − IGESKnot[i + 1]);
  backward[i] = 1./(1. − l) * _ctrl_pts[i + 1] + (1. − 1./(1. − l)) * backward[i + 1];
}
for (long i = r − k + v − 2; i ≥ 0; i−−) {
  backward[i] = _ctrl_pts[i];
}
```

이 코드는 184번 마디에서 사용된다.

188. ⟨ Blend forward and backward control points 188 ⟩ ≡

```
for (size_t i = r - k + v - 1; i <= r - 1; i++) {
  double mu = get_blending_ratio(IGESKnot, v, r, i);
  _ctrl_pts[i] = (1. - mu) * forward[i] + mu * backward[i];
}
for (size_t i = r; i <= _ctrl_pts.size() - 2; i++) {
  _ctrl_pts[i] = _ctrl_pts[i + 1];
}
_ctrl_pts.pop_back();
```

이 코드는 184번 마디에서 사용된다.

189. ⟨ Methods of **cubic_spline** 103 ⟩ +≡

```
public:
  void remove_knot(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 ⟩ ≡

```
  void cubic_spline::write_curve_in_postscript(
          psf &ps_file,
          unsigned dense,
          float line_width,
          int x, int y,
          float magnification
          ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    ps_file << "newpath" << endl << "[] 0 setdash " << line_width << " setlinewidth" << endl;
    point pt(magnification * evaluate(_knot_sqnc[2], 2));
    ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
    double incr = (_knot_sqnc[_knot_sqnc.size() - 3] - _knot_sqnc[2])/double(dense);
    for (size_t i = 0; i != dense + 1; i++) {
      double u = _knot_sqnc[2] + incr * i;
      pt = magnification * evaluate(u);
      ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
    }
 #if 0
    for (size_t i = 2; i < _knot_sqnc.size() - 3; i++) {
      if (_knot_sqnc[i] < _knot_sqnc[i + 1]) {
        double knot = _knot_sqnc[i];
        double incr = (_knot_sqnc[i + 1] - knot)/double(dense);
        double u = knot;
        for (size_t j = 0; j <= dense; j++) {
          pt = magnification * evaluate(u, i);
          ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
          u += incr;
        }
      }
    }
 #endif
    ps_file << "stroke" << endl;
    ps_file.flags(previous_options);
  }
  void cubic_spline::write_control_polygon_in_postscript(
```

```
            psf &ps_file,         float line_width,
            int x, int y,
            float magnification
            ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    ps_file << "newpath" << endl << "[] 0 setdash " << .5 * line_width << " setlinewidth" << endl;
    point pt(magnification * _ctrl_pts[0]);
    ps_file << pt(x) << "\t" << pt(y) << "\t" << "moveto" << endl;
    for (size_t i = 1; i < _ctrl_pts.size(); i++) {
        pt = magnification * _ctrl_pts[i];
        ps_file << pt(x) << "\t" << pt(y) << "\t" << "lineto" << endl;
    }
    ps_file << "stroke" << endl;
    ps_file.flags(previous_options);
}
void cubic_spline::write_control_points_in_postscript(
            psf &ps_file,
            float line_width,
            int x, int y,
            float magnification
            ) const {
    ios_base::fmtflags previous_options = ps_file.flags();
    ps_file.precision(4);
    ps_file.setf(ios_base::fixed, ios_base::floatfield);
    point pt(magnification * _ctrl_pts[0]);
    ps_file << "0 setgray" << endl << "newpath" << endl << pt(x) << "\t" << pt(y) << "\t" <<
        (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl << "closepath" <<
        endl << "fill stroke" << endl;
    if (_ctrl_pts.size() > 2) {
        for (size_t i = 1; i <= (_ctrl_pts.size() - 2); i++) {
            pt = magnification * _ctrl_pts[i];
            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;
        }
        pt = magnification * _ctrl_pts.back();
        ps_file << "0 setgray" << endl << "newpath" << endl << pt(x) << "\t" << pt(y) << "\t" <<
            (line_width * 3) << "\t" << 0.0 << "\t" << 360 << "\t" << "arc" << endl << "closepath" <<
            endl << "fill stroke" << endl;
```

```
        }
        ps_file.flags(previous_options);
    }
```

이 코드는 101번 마디에서 사용된다.

191. ⟨Methods of **cubic_spline** 103⟩ +≡

```
public:
    void write_curve_in_postscript(
            psf &, unsigned, float, int x = 1, int y = 1,
            float magnification = 1.0) const;
    void write_control_polygon_in_postscript(
            psf &, float, int x = 1, int y = 1,
            float magnification = 1.0) const;
    void write_control_points_in_postscript(
            psf &, float, int x = 1, int y = 1,
            float magnification = 1.0) const;
```

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

Computer-Aided Geometric Design

(Last revised on October 28, 2016)