About **RMCP**Library

by
Syung-Kwon Ra
(syungkwon.ra@gmail.com)
Center for Cognitive Robotics
Korea Institute of Science and Technology

February 5, 2009

Contents

List of Figures

List of Tables

Change Log

버전	일시	고친이	내용
최초작성, 0.01	2006-04-05	나성권	
0.1	2009-02-05	나성권	

Chapter 1

Introduction

1.1 Objective

RMCP(Ra Mathematics C++) Library는 로봇 연구에 필요한 수학 연산 함수들을 모아 놓은 C++ 라이브러리 이다. 일반적인 행렬에 대한 클래스(Matrix)와 Lie Group과 Lie Algebra에 속하는 행렬에 대한 클래스(so3, SO3, se3, SE3), 그리고 이와 연관되어 사용되는 행렬에 대한 클래스(Inertia, AInertia)을 C++로 구현하였다. ISO/ANSI C++ 규격을 준수하였으며, 표준 라이브러리에만 기반하도록 설계하였다. 모든 소스코드는 Visual C++ .NET 2005(윈도우), GCC(G++) 4.0(리눅스)에서 테스트하였다. 내부적으로 Intel Math Kernel Library(이하 MKL)를 사용하고 있다.

1.2 Prerequisite

RMCP은 C++ Programming Language로 작성되었다. 원활한 사용을 위해서 C++에 대한 이해 이외에 아래와 같은 기초지식이 요구된다.

추가 예정 추가.

1.3 Directory

RMCP의 디렉토리 구조.

```
[rmcp]
  |---[doc]
  |---[include]
  |      +---[rmcp]
  |---[lib]
  |---[linux]
  |      +---[rmcp_lib]
  |      +---[rmcp_test]
  |---[src]
```

```
|---[vender]
|       +---[linux]
|               +---[include]
|               +---[lib]
|       +---[windows]
|               +---[include]
|               +---[lib]
+---[win32]
        +---[rmcp_lib]
        +---[rmcp_test]
```

/doc 문서들.

/include 헤더파일들

/lib 정적라이브러리 파일. 컴파일의 결과로 생성된다.

/linux 리눅스에서의 컴파일용 make file 등.

/src 소스 파일들.

/vender **RMCP**가 의존하는 MKL이 들어 있다. linux 디렉토리에는 mkl이 linux 버전이, windows 디렉토리에는 windows 버전이 있다.

/win32 윈도우에서의 컴파일용 sln file 등. 이 폴더안의 rmcp.sln을 클릭하면 **RMCP**을 컴파일하고, 이를 이용하는 예제를 볼 수 있다.

Chapter 2

Classes

RMCP에서 제공하는 클래스 이름은 대부분 **?**====** 의 형태로 되어 있다. 맨뒤에 위치하는 ====는 클래스의 대분류를 나타내며, 예를 들어 vector, matrix, spline가 같은 단어가 위치한다. 중간에 있는 **는 두글자의 알파벳이며, 소분류가 된다. 예를 들어 symatrix 는 symmetric matrix, trmatrix는 triangular matrix를 의미한다. 클래스이름은 s, d, c, z중의 하나로 시작하는데, 이것은 클래스의 데이터 타입을 나타내며 ?문자로 표기된다. s, d, c, z가 의미하는 바는 표 2.1에 나타나 있다.

	의미	data type
s	single-precision real number	float
d	double-precision real number	double
c	single-precision complex number	
z	double-precision complex number	

Table 2.1: s, d, c, z

클래스의 대분류에 대해서 정리하면 아래와 같다.

Vector Class 벡터에 관한 클래스이며, ?**vector의 이름을 갖는다.

Matrix Class 행렬에 관한 클래스이며, ?**matrix의 이름을 갖는다.

Derived Vector, Matrix Class 벡터와 행렬 클래스에서 파생된 특수용도의 벡터이다.

Optimization Class 최적화 수치해석을 위한 클래스이며, ?**optimize의 이름을 갖는다.

Spline Class 곡선에 관한 클래스이며, ?**spline의 이름을 갖는다.

2.1 Vector Class

벡터 클래스는 아래와 같은 공통적인 연산과 멤버함수를 제공한다.

2.2 Matrix Class

행렬 클래스는 아래와 같은 공통적인 연산과 멤버함수를 제공한다.

2.3 Derived Vecotr, Matrix Class

2.3.1 Robotics

2.3.2 Lie Group

2.4 Optimization Class

2.5 Spline Class

2.6 상수

RMCP에서 사용하는 상수는 `rmcp.h`에 아래와 같이 정의되어 있다.

1. `PI = 3.14159265358979323846264338327950288419`7 π
2. `PI_2 = 1.57079632679489661923132169163975144209`9 $\pi/2$
3. `MATRIX_NORM_EPS = 1.0E-20` matrix 계열 class
4. `VECTOR_NORM_EPS = 1.0E-20` vector 계열 class
5. `EXP_EPS = 1.0E-12` lie group class
6. `LOG_EPS = 1.0E-12` lie group class
7. `INV_EULER_EPS = 1.0E-16` lie group class
8. `SQOPTIMIZE_INF = 1.0E20` dsqoptimize class

Chapter 3

Install

이 장에서는 **RMCP**을 사용하기 위해서 컴파일 환경을 어떻게 구축해야 하는지에 대해서 다룬다.

3.1 Windows

윈도우 환경에서는 Visual Studio .NET 2005을 기준으로 설명한다. 윈도우에서는 MKL은 따로 설치할 필요가 없다.

1. 그림 3.1과 같이 `my_program` 폴더에 있는 program1 솔루션에서 **RMCP**을 사용하고자 한다고 하자.

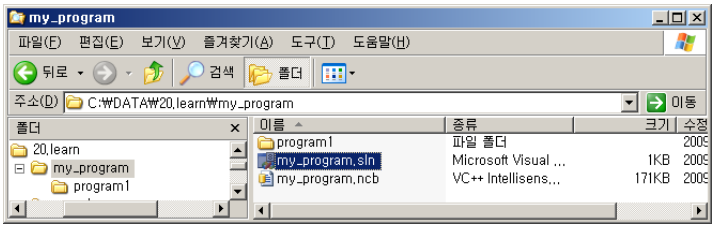

Figure 3.1: my program folder

2. **RMCP**와 **RMCP**가 의존하고 있는 MKL을 `my_program`에서 사용하기 위하여 복사해온다. `my_program` 폴더에 vender 폴더를 만들고, vender 폴더에 include 폴더와 lib 폴더를 만든다.

3. **RMCP**에서 `/include` 안에 있는 rmcp 폴더를 통채로 `my_program/vender/include` 폴더 안에 복사한다. `/include/rmcp` 폴더 안의 헤더 파일들만 복사하지 말고 반드시 `/include/rmcp` 폴더를 통채로 복사한다. 그리고, `/vender/windows/include` 폴더 안에서 mkl 폴더를 통채로 `my_program/vender/include` 폴더 안에 복사한다. 결과적으로, `my_program/vender/include` 폴더 안에는 rmcp 폴더와 mkl 폴더가 생기게 된다.

4. **RMCP**에서 `/lib` 폴더 안에서 rmcp.lib 과 rmcpD.lib 파일을 `my_program/vender/lib` 폴더 안에 복사한다. 그리고, **RMCP**에서 `/vender/windows/lib` 안의 .lib 파일들을 역시 같은 곳으로 복사한다.

5. 이제 파일복사는 끝났다. 이제 아래 그림 3.2과 같은 폴더 구조를 가지게 된다.

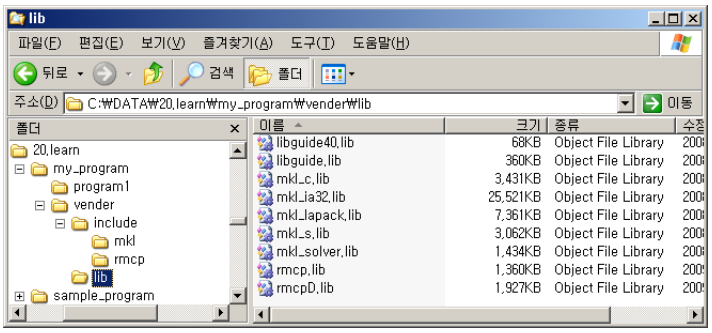

Figure 3.2: **RMCP**을 복사 완료한 후의 my program folder

6. `my_proram/program.sln`을 클릭하여 Visual Studio .NET 2003 을 실행한다. (그림 3.3)

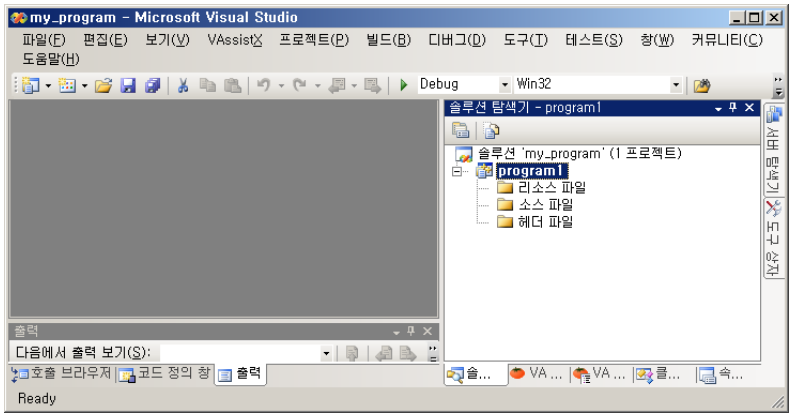

Figure 3.3: Visual Studio

7. 오른쪽의 "솔루션 탐색기"에서 "program1" 위치에서 마우스 오른쪽을 클릭하여 "속성"를 선택한다. 그러면, "속성 페이지"가 화면에 뜬다. "속성 페이지"의 왼쪽 상단에 있는 "구성"에서 "모든 구성"으로 바꿔준다. 이것은 이후 작업할 설정 내용들이 Debug와 Release 모드 모두에 적용되게 하기 위함이다. (그림 3.4)

8. 왼쪽 트리모양의 항목에서 "구성 속성 > C/C++ > 일반"을 찾아간 후, 추가 포함 디렉터리 항목에 `./;../vender/include/;`를 적어준다. 이 경로는 소스 파일이 있는 곳으로 부터의 상대경로이다.(그림 3.5)

9. 왼쪽 트리모양의 항목에서 "구성속성 > 링커 > 일반"을 찾아간 후, 추가 라이브러리 디렉터리 항목에 "../vender/lib/;"을 적어준다. 이 경로는 program1.vcproj 파일이 있는 곳으로 부터 상대경로이다.(그림 3.6)

10. 왼쪽 트리모양의 항목에서 "구성속성 > 링커 > 입력"을 찾아간 후, 추가 종속성 항목에 `mkl_c.lib libguide.lib rmcp.lib`을 적어준다. 앞의 두 .lib 파일은 MKL이고, 뒤의 rmcp.lib 파일은 **RMCP**라이브러리이다. (그림 3.7)

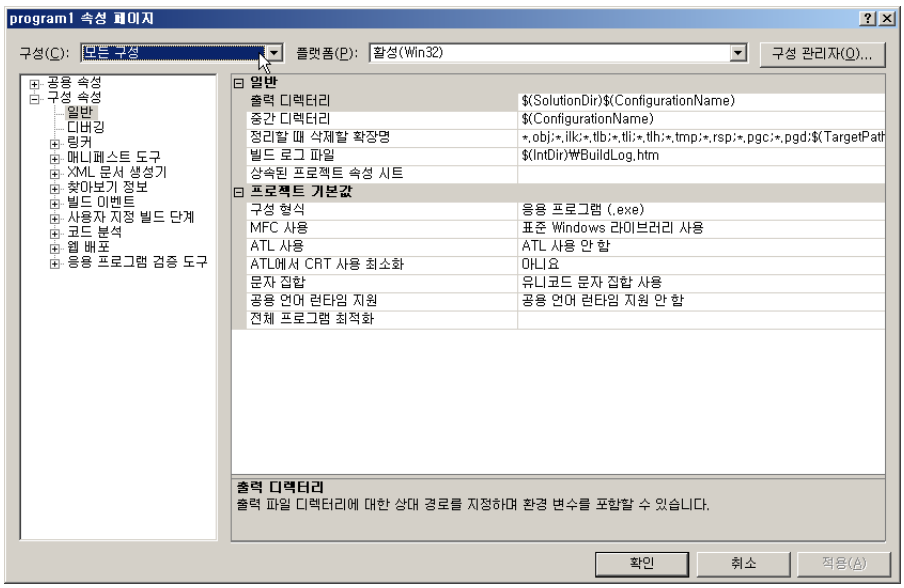

Figure 3.4: 속성 페이지에서 모든 구성으로 바꿔준다.

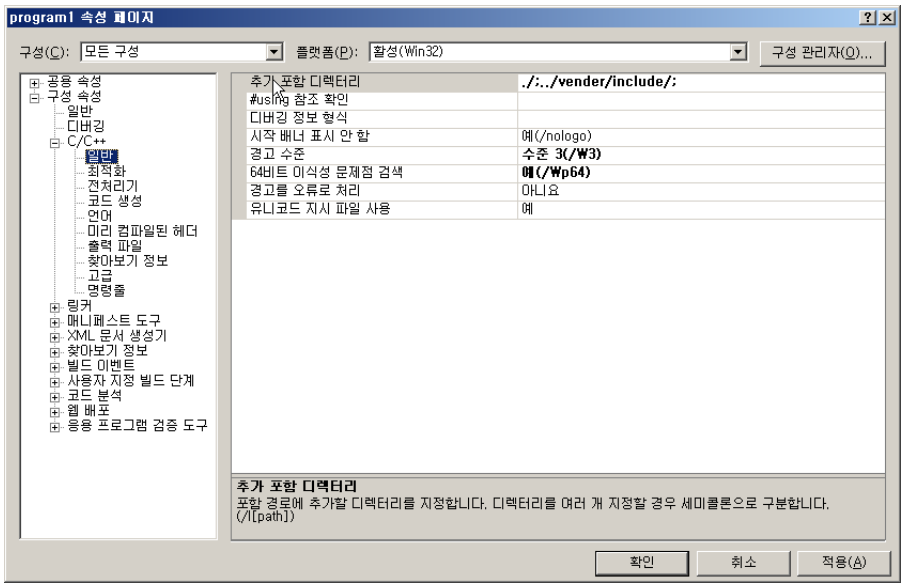

Figure 3.5: 헤더 파일의 경로를 설정해 준다.

11. 왼쪽 상단의 "구성"을 "Debug"로 바꾼후 "추가 종속성"항목에 적은 rmcp.lib을 rmcpD.lib으로 바꿔준다. 이 부분은 라이브러리의 기능에는 영향이 없고, 단지 디버그를 편리하게 할수 있도록 하기 위함이다. 편의상 이 과정은 건너뛰어도 된다. (그림 3.8)

12. 확인을 누르고 속성 페이지를 빠져나와서 프로그래밍을 진행하면 된다. 그림 3.9, 3.10는 **RMCP**중에서 dvector와 dmatrix의 곱을 하는 예이다.

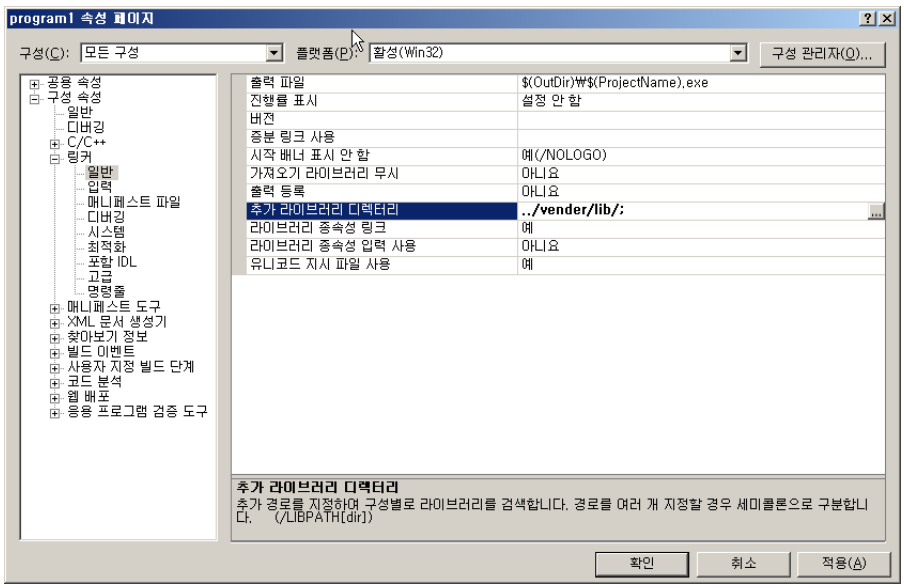

Figure 3.6: 정적 라이브러리 파일의 경로를 추가해준다.

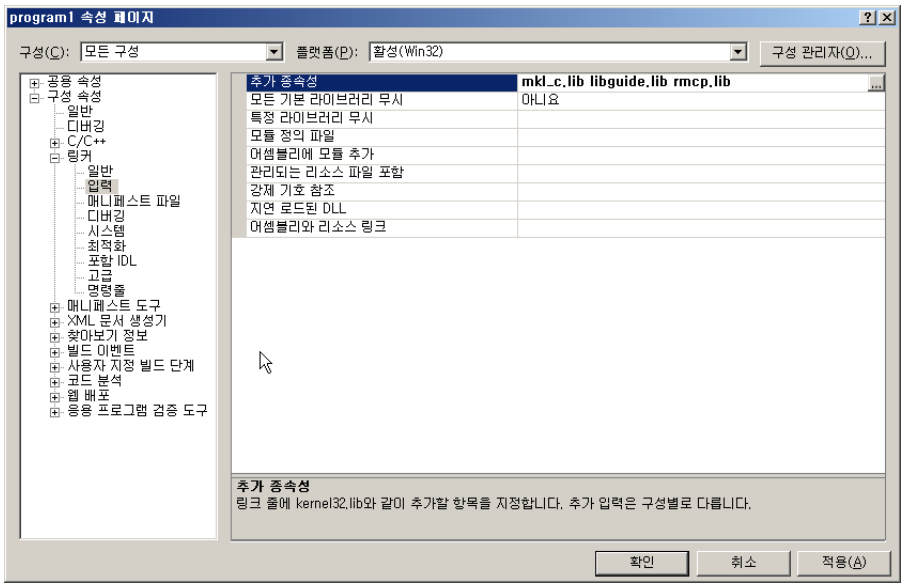

Figure 3.7: 사용할 정적 라이브러리 파일를 지정해준다.

3.2 Linux

리눅스에서는 MKL의 리눅스 버전을 설치해야 한다. 설치후에 .bashrc 파일에 아래와 같이 추가해준다.

```
export LD_LIBRARY_PATH=/usr/xenomai/lib:/opt/intel/mkl/8.0/lib/32:$LD_LIBRARY_PATH
```

그리고 Makefile을 작성할때는 아래와 같은 예를 참고하라

```
LIB = ../lib
```

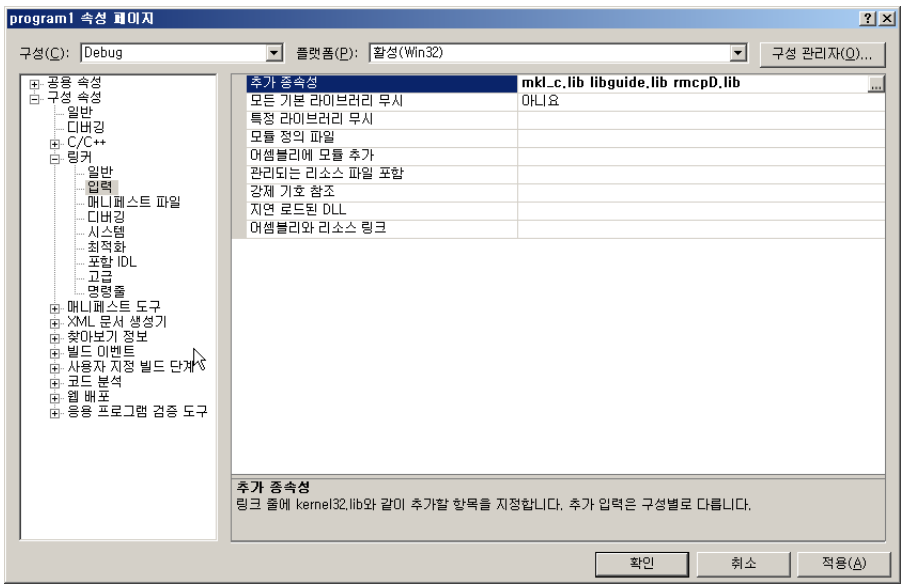

Figure 3.8: 디버그 모드에서 사용할 정적 라이브러리 파일를 지정해준다.

```
SRC = .
CC = g++

CFLAGS = -O2
INC = -I../include -I../vender/linux/include
LIB = -L../lib -L../vender/linux/lib

SRC_FILES = $(SRC)/util.cpp \
            $(SRC)/main.cpp

LIB_FILES = -lm -lrmcp -lmkl_ia32 -lmkl_lapack -lguide -lmkl -lmkl_solver

all : lib test

test : $(SRC_FILES)
     $(CC) -o test $(CFLAGS) $(INC) $(LIB) $(SRC_FILES) $(LIB_FILES)

clean :
     rm -f test *.o
```

기억해야 할 것은 헤더 파일은 C/C++ 소스 파일이 있는 곳에서의 상대경로를 지정해 주고, 정적 라이브러리 파일은 Makefile 이 있는 곳에서의 상대경로를 지정해 주어야 한다는 것이다.

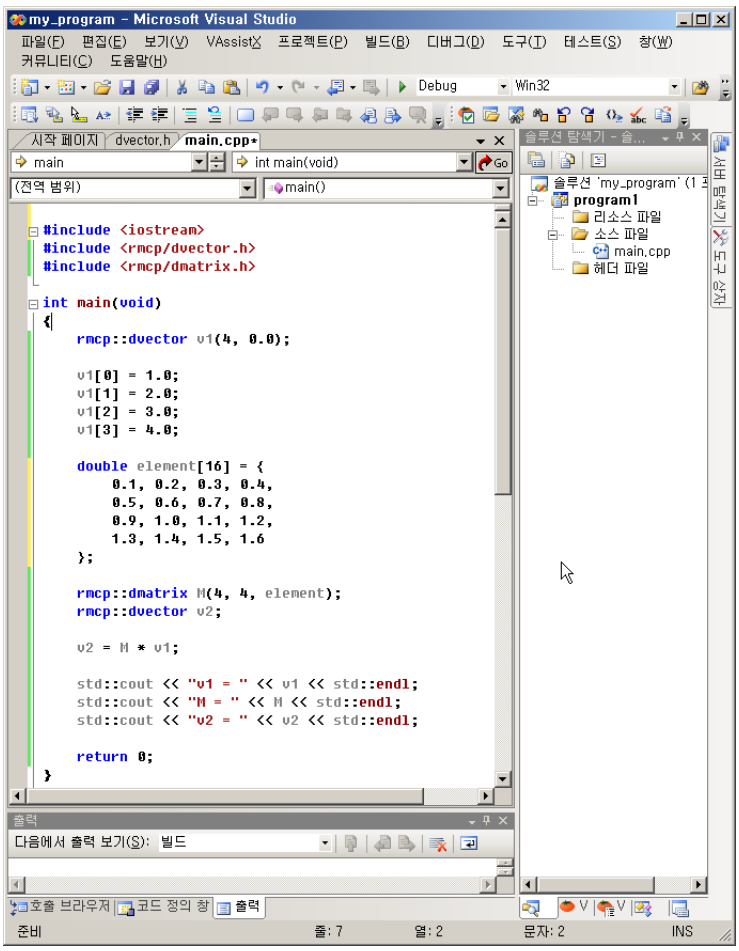

Figure 3.9: rmcp 사용예

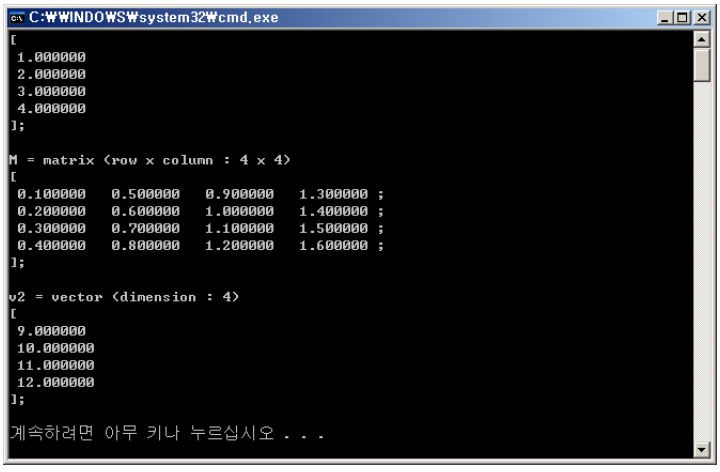

Figure 3.10: rmcp 사용예의 결과

Chapter 4

Optimization

현재 **RMCP**에서 제공하는 최적화 방법은 sequential quadric programming 을 구현한 `dsqoptimize` Class 1가지 이다.

4.1 dsqoptimize Class

`dsqoptimize` class는 C 언어로 프로그래밍 된 donlp2[1] 라이브러리에기반을 두고 C++에 적합한 인터페이스를 구현한 것이다. dsqoptimize class의 내부적인 알고리즘은 donlp2를 그대로 사용하고 있으므로, donlp2에 제공되는 메뉴얼[2]을 참고한다면 도움이 될 것이다. `dsqoptimize` class는 `dsqobjectfn` class와 같이 사용해야 하며, 사용순서는 아래와 같다.

1. `dsqobjectfn` class을 상속받아서 목적함수를 나타내는 클래스를 만든다. 이 클래스는 아래와 같은 함수를 재정의(overriding) 해야 한다. 만약 해당 함수가 필요가 없다면, 내용이 없는 빈 함수로써 재정의한다.

   - `double evaluate(double x[])` : 변수 x를 입력받아 목적함수의 값을 리턴한다.
   - `gradient(double x[], double grad[])` : 목적함수의 구배가 해석적으로 존재한다면, 변수 x를 입력받아서 구배값을 grad에 담아 리턴한다. 이 함수를 재정의하였다면, dsqoptimize class 사용시에 `set_analytic(true)`를 하여주어야 한다. 그렇지 않다면, dsqoptimize class가 목적함수의 구배를 수치적으로 구할수 있도록 `set_analytic(false)`를 하여야 한다.
   - `nonlinear_constraint(double x[], double con[], int err[])` : 비선형 제약조건을 정해주는 첫번째 함수이다. 변수 x를 받아서 제약조건의 값을 con에 넣어준다. 이 때, 여러 제약조건의 계산값을 con 배열에 모두 넣어준다. 만약 어떤 특정한 x에 대해서는 제약조건을 계산할수 없는 경우에는(0으로 나누는 경우 등) err를 1로 지정해주면 된다. 이 경우에는 최적화 과정에서 current correction step의 크기를 줄여가면서 특정 x를 회피하려는 시도를 하게된다.
   - `nonlinear_constraint(double x[], int i, double &con, int &err)` : 비선형 제약조건을 정해주는 두번째 함수이다. 이 두번째 함수는 첫번째 함수와 달리, i번째 함수의 값만 con에 넣어주면 된다. 비선형 제약조건은 첫번째와 두번째 형태 모두 구현을

하여야 하는데, 비선형 제약조건을 이처럼 두 종류의 함수로 중복해서 구현하는 이유는 최적화시 계산의 효율성을 기하기 위해서이다.

- `nonlinear_gradient(double x[], int i, double grad[])` : 비선형 제약조건에 대해서 구배함수를 지정한다. 변수 x에 대하여 i번째 비선형 제약조건에 대한 구배값을 grad에 넣어준다.

2. 위에서 dsqobjectfn을 상속받은 class의 객체와 dsqoptimize class의 객체를 하나 만든다. dsqoptimize class는 생성자로써 dsqobjectfn을 상속받은 class의 객체주소를 인자로 받는다.

3. dspoptimize class의 객체를 초기화 한다.

   - `init(int dim, int linear, int nonlinear, int iteration, int step)` dim은 변수 x의 개수, linear는 선형제약조건의 개수, nonlinear는 비선형 제약조건의 개수이다. iteration과 step은 지정하지 않아도 되며, 지정하지 않으면 기본적으로 100, 20의 값을 각각 사용한다. iteration과 step을 크게 지정할수록 최적화 시간이 오래 소요되나, 정밀한 값을 얻을수 있다.

4. 변수 x와 선형제약조건, 비선형 제약조건의 최대값과 최소값을 설정한다. 최대값과 최소값이 없는 경우, 지정하지 않아도 무방하며 이경우에는 최대값과 최소값으로 각각 `SQOPTIMIZE_INF`와 `-SQOPTIMIZE_INF`가 사용된다. equality constraint의 경우에는 최대값과 최소값에 같은 값을 넣어주면 된다.

   - `set_upper(int i, double value)`, `set_lower(int i, double value)` i번째 x의 최대, 최소값 지정
   - `set_lincon_upper(int i, double value)`, `set_lincon_lower(int i, double value)` i번째 선형 제약조건의 최대, 최소값 지정
   - `set_nonlincon_upper(int i, double value)`, `set_nonlincon_lower(int i, double value)` i번째 비선형 제약조건의 최대, 최소값 지정

5. linear constraint가 있으면, dmatrix의 형태로 설정해준다.

   - `set_linear_constraint(dmatrix &constraint)`

6. 최적화 수행을 위한 여러 상태변수들을 설정한다.

   - `set_analytic(bool analytic)` 해석적인 구배함수를 제공하면 true, 아니면 false
   - `set_silent(bool silent)` 최적화 과정을 로그파일로 출력할 것이면 false, 아니면 true
   - `set_logfile(string name)` 출력할 로그파일의 이름지정
   - `set_difftype(int difftype)`
   - `set_tau0(double tau0)`
   - `set_del0(double del0)`
   - `set_nreset(int nreset)`
   - `set_epsdif(double epsdif)`
   - `set_epsfcn(double epsfcn)`

- `set_taubnd(double taubnd)`
- `set_te0(bool te0)`
- `set_te1(bool te1)`
- `set_te2(bool te2)`
- `set_te3(bool te3)`

7. 최적화를 수행한다

   - `solve(dvector initial_x)` initial value를 solve 함수의 인자로 넘겨주어 최적화를 실행한다.

8. 최적화 수행 결과를 얻어낸다.

   - `dvector optimal_solution()` 최적 x값이 dvector로 리턴된다.
   - `double optimal_value()` 최적값이 리턴된다.
   - `double elapsed_time()` 최적화에 소요된 시간이 초단위로 리턴된다.

4.1.1 example 1 : simple

The problem is to minimize

$$f(x) = x_0^2 + x_1^2, \quad x = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \in \mathbf{R}^2 \tag{4.1}$$

$$\nabla f(x) = \begin{bmatrix} 2x_0 \\ 2x_1 \end{bmatrix} \tag{4.2}$$

subject to

$$g(x) = x_0 \times x_1 = 5.0 \tag{4.3}$$

$$\nabla g(x) = \begin{bmatrix} x_1 \\ x_0 \end{bmatrix} \tag{4.4}$$

and the bounds

$$-100 \leq x_0 \leq 100 \tag{4.5}$$

$$-100 \leq x_1 \leq 100 \tag{4.6}$$

```
#include "rmcp/dsqobjectfn.h"
class simple : public rmcp::dsqobjectfn {
public:
    double evaluate(double x[]);
    void gradient(double x[], double grad[]);
    void nonlinear_constraint(double x[], double con[], int err[]);
    void nonlinear_constraint(double x[], int i, double &con, int &err);
    void nonlinear_gradient(double x[], int i, double grad[]);
```

```
};
// 목적함수
double
simple::evaluate(double x[])
{
    return x[0] * x[0] + x[1] * x[1];
}
// 구배함수
void
simple::gradient(double x[], double grad[])
{
    grad[0] = 2.0 * x[0];
    grad[1] = 2.0 * x[1];
}
// 비선형 제약조건
void
simple::nonlinear_constraint(double x[], double con[], int err[])
{
    con[0] = x[0] * x[1];
}
// 비선형 제약조건
void
simple::nonlinear_constraint(double x[], int i, double &con, int &err)
{
    if (i == 0) {
        con = x[0] * x[1];
    }
}
// 비선형 제약조건에 대한 구배함수
void
simple::nonlinear_gradient(double x[], int i, double grad[])
{
    if (i == 0)  {
        grad[0] = x[1];
        grad[1] = x[0];
    }
}

void simple_test()
{
simple fn;
```

```
rmcp::dsqoptimize OPT(&fn);
OPT.init(2, 0, 1, 4000, 20);

OPT.set_upper(0, 100.0);
OPT.set_upper(1, 100.0);
OPT.set_lower(0, -100.0);
OPT.set_lower(1, -100.0);
OPT.set_nonlincon_upper(0, 5.0);
OPT.set_nonlincon_lower(0, 5.0);

OPT.set_analytic(true);
OPT.set_silent(false);
OPT.set_logfile("simple2");
OPT.set_epsdif(1.e-16);
OPT.set_epsfcn(1.e-16);
OPT.set_taubnd(1.0);
OPT.set_del0(1.0e0);
OPT.set_tau0(1.e1);
OPT.set_te2(true);
OPT.set_te3(true);

rmcp::dvector initial_x(2, 10.0);

OPT.solve(initial_x);

rmcp::dvector optimal(2);

optimal = OPT.optimal_solution();

std::cout << optimal << std::endl;
std::cout << OPT.optimal_value() << std::endl;

std::cout << OPT.elapsed_time() << std::endl;
}
```

4.1.2 example 2 : fmincon

4.1.3 example 3 : portfol1

4.1.4 example 4 : hs114new

Appendix A

Intel Math Kernel Library

intel mkl의 설명이 들어갈 부분.

Bibliography

[1] http://plato.asu.edu/donlp2.html

[2] DONLP2-INTV-DYN USERS GUIDE, P.Spellucci