2009 년 2 학기 전산선박설계

Programming Assignment #4

SQP Method

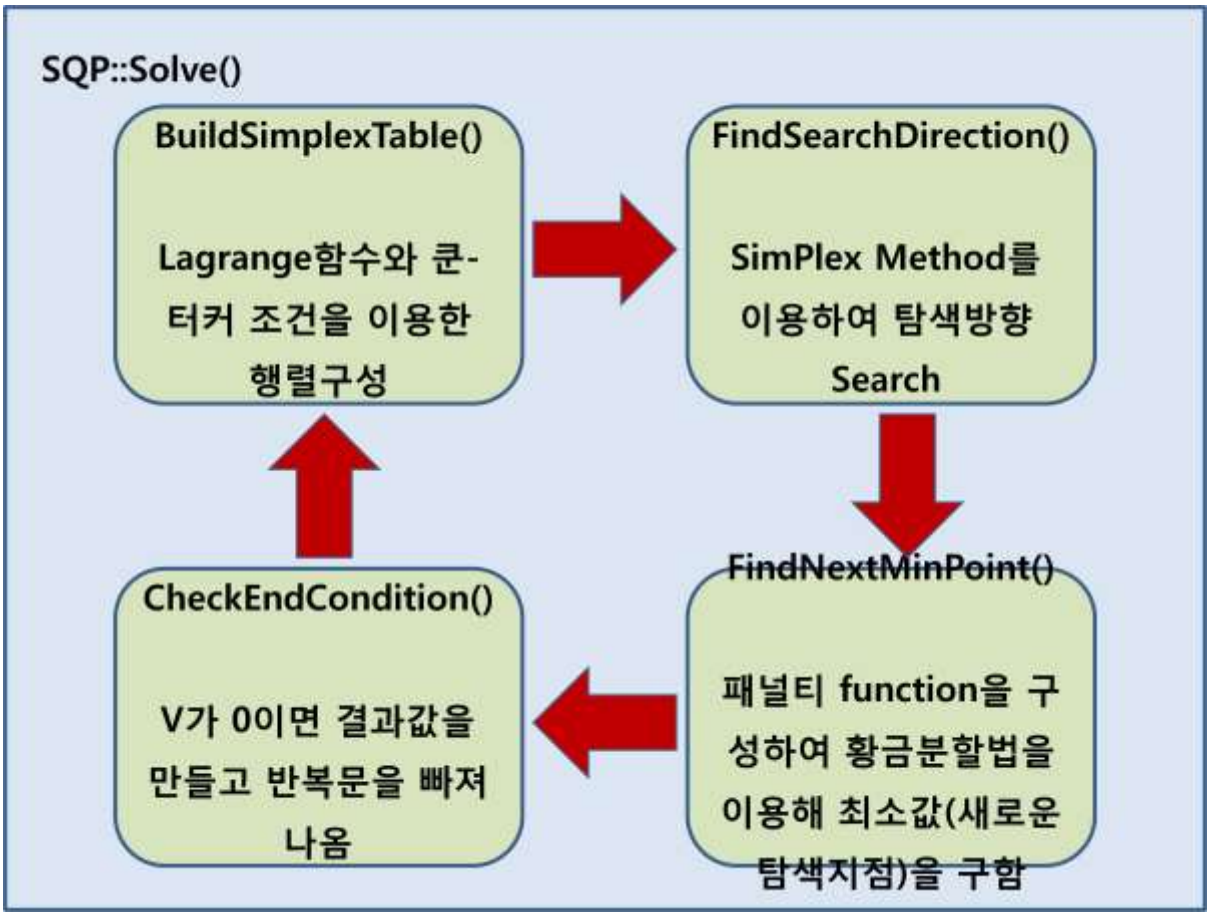
2005-11934 김재현

2007-11911 배상준

2009-10-25

목차

1. SQP Method 알고리즘



이번 프로젝트에서의 목적은 SQP method를 이용하여 어떤 목적함수와 그에 대한 여러 제약조건들이 있을 때 목적함수의 최소값을 제약조건을 만족시키면서 구하는 것이다. SQP method는 지난 번에 했던 여러 방법들을 포괄하여 만든다. 먼저 1차 프로젝트에서 수행했던 황금분할법, 그리고 바로 이전에 했던 Simplex method를 이용한다. Simplex method는 어떤 목적함수를 근사시킨후 이 근사시킨 목적함수에서의 최소값의 방향을 탐색하는데 이용하게 되고 황금분할법은 탐색시작점과 탐색 방향이 전과정에서 구해지면 이를 이용해서 최소점(국소적 최소점)을 구하게 된다. 그리고 이 과정은 계속 반복되면서 목표하는 최소값을 구한다. 위의 알고리즘이 SQP 클래스에 있는 Solve함수인데 이 과정을 잘 보여주고 있다.

```
void SQP::Solve()
{

    while(1)
    {

        BuildSimplexTable(); //완성
        FindSearchDirection();
        FindNextMinPoint();

        if(CheckEndCondition() == true)
        {
            m_MinValue = m_ObjFn(m_x);
            break;
        }


    }

}
```

2. Simplex table 구성

$$\begin{bmatrix} \mathbf{H}_{(n\times n)} & -\mathbf{H}_{(n\times n)} & \mathbf{A}_{(n\times m)} & \mathbf{0}_{(n\times m)} & \mathbf{N}_{(n\times p)} & -\mathbf{N}_{(n\times p)} \\ \mathbf{A}^T{}_{(m\times n)} & -\mathbf{A}^T{}_{(m\times n)} & \mathbf{0}_{(m\times m)} & \mathbf{I}_{(m\times m)} & \mathbf{0}_{(m\times p)} & \mathbf{0}_{(m\times p)} \\ \mathbf{N}^T{}_{(p\times n)} & -\mathbf{N}^T{}_{(p\times n)} & \mathbf{0}_{(p\times m)} & \mathbf{0}_{(p\times m)} & \mathbf{0}_{(p\times p)} & \mathbf{0}_{(p\times p)} \end{bmatrix} \begin{bmatrix} \mathbf{d}^+{}_{(n\times 1)} \\ \mathbf{d}^-{}_{(n\times 1)} \\ \mathbf{u}_{(m\times 1)} \\ \mathbf{s}_{(m\times 1)} \\ \mathbf{y}_{(p\times 1)} \\ \mathbf{z}_{(p\times 1)} \end{bmatrix} = \begin{bmatrix} -\mathbf{c}_{(n\times 1)} \\ \mathbf{b}_{(m\times 1)} \\ \mathbf{e}_{(p\times 1)} \end{bmatrix} = \mathbf{D}_{((n+m+p)\times 1)}$$

$$= \mathbf{B}_{((n+m+p)\times(2n+2m+2p))} \qquad = \mathbf{X}_{((2n+2m+2p)\times 1)}$$

$$u_i s_i = 0; i = 1 \text{ to } m$$

$$\mathbf{B}_{((n+m+p)\times(2n+2m+2p))}\mathbf{X}_{((2n+2m+2p)\times 1)} = \mathbf{D}_{((n+m+p)\times 1)}$$

B행렬은 H, A, N, I 행렬로 구성되어있다.

- H : 개념상으로는 Hessian Matrix 이지만 여기에선 단위행렬이라고 알면 될듯하다.
- A : 부등호 제약조건의 gradient 값이다. (n행 m열)
- N : 등호 제약조건의 gradient 값이다. (n행 p열)
- I : 단위행렬. (m행 m열)

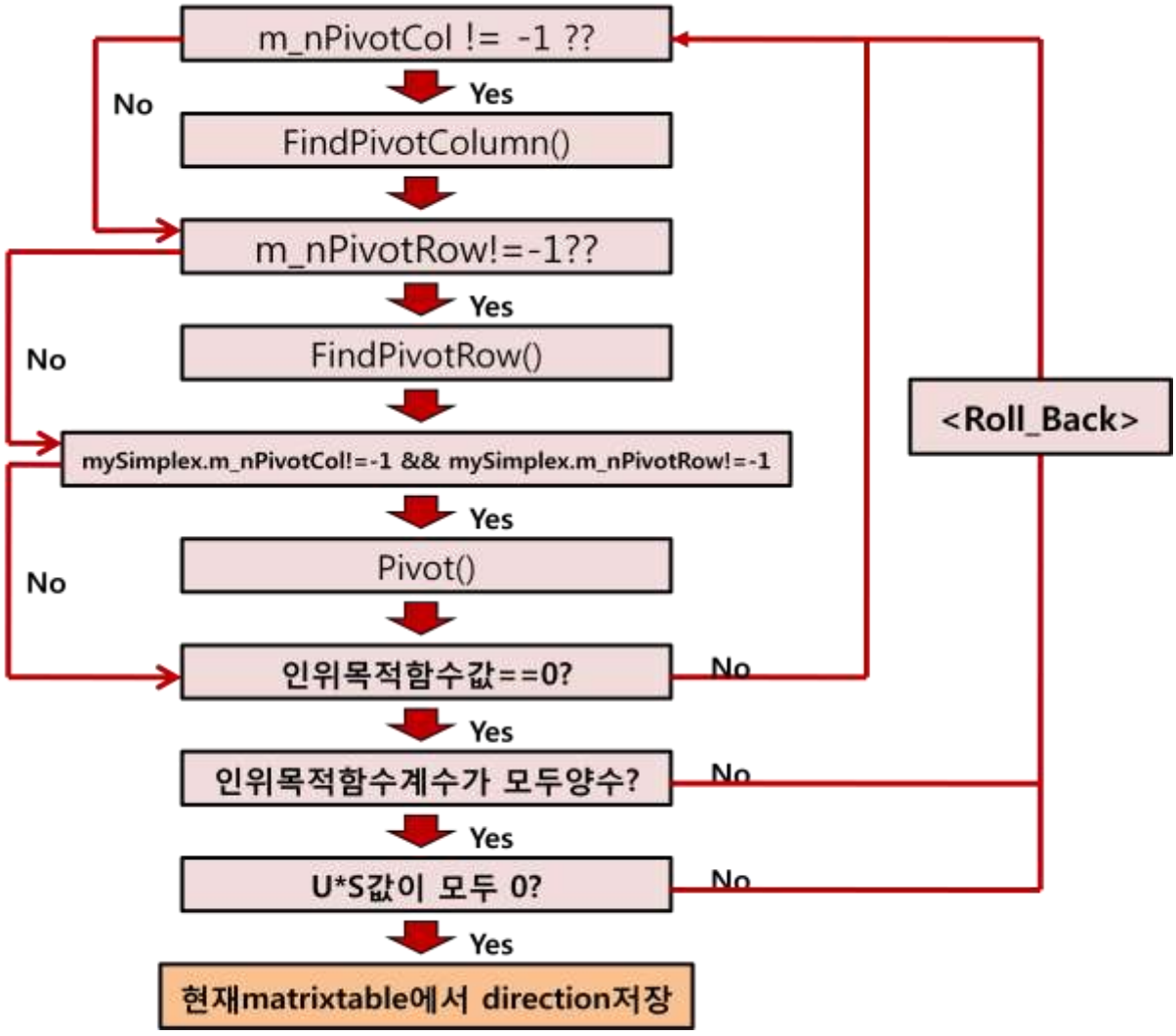
D행렬은 c, b, e 행렬로 구성되어있다.

- c : 목적함수의 gradient 값이다. (n행 1열)
- b : 부등호 제약조건의 함수 값 × (-1) (m행 1열)
- e : 등호 제약조건의 함수 값 × (-1) (p행 1열)

$$\left[\begin{array}{cccccc|ccc|c} \mathbf{H}_{(n\times n)} & -\mathbf{H}_{(n\times n)} & \mathbf{A}_{(n\times m)} & \mathbf{0}_{(n\times m)} & \mathbf{N}_{(n\times p)} & -\mathbf{N}_{(n\times p)} & \mathbf{I}_{(n\times n)} & \mathbf{0}_{(n\times m)} & \mathbf{0}_{(n\times p)} & -\mathbf{c}_{(n\times 1)} \\ \mathbf{A}^T{}_{(m\times n)} & -\mathbf{A}^T{}_{(m\times n)} & \mathbf{0}_{(m\times m)} & \mathbf{I}_{(m\times m)} & \mathbf{0}_{(m\times p)} & \mathbf{0}_{(m\times p)} & \mathbf{0}_{(m\times n)} & \mathbf{I}_{(m\times m)} & \mathbf{0}_{(m\times p)} & \mathbf{b}_{(m\times 1)} \\ \mathbf{N}^T{}_{(p\times n)} & -\mathbf{N}^T{}_{(p\times n)} & \mathbf{0}_{(p\times m)} & \mathbf{0}_{(p\times m)} & \mathbf{0}_{(p\times p)} & \mathbf{0}_{(p\times p)} & \mathbf{0}_{(p\times n)} & \mathbf{0}_{(p\times m)} & \mathbf{I}_{(p\times p)} & \mathbf{e}_{(p\times 1)} \end{array}\right]$$

Artificial Object Function
(모든 열의 합에 (-)를 붙여 구함)

3. 탐색방향 Search



- FindSearchDirection 함수안에는 위와 같은 알고리즘으로 함수 코드가 구현되어 있다. 차근차근 알고리즘을 분석해보자. 전체적으로 while(1)문으로 구성되어 있어 조건이 하나라도 만족하지 못하면 무한 반복하게 되어 있다. 그러나 무한 반복할때는 항상 새로운 메트릭스를 반복시키게 되

```
    Simplex* temp = new Simplex;
    temp = mySimplex.m_vSimplexStack[mySimplex.m_vSimplexStack.size()-1];
    mySimplex.m_vSimplexStack.pop_back();
    cout<<"피벗전"<<endl;
    temp->m_matSimplexTable.PrintElementConsole();
    temp->Pivot();
    cout<<"피벗후"<<endl;
    temp->m_matSimplexTable.PrintElementConsole();
    //mySimplex=test;
    mySimplex.m_matSimplexTable = temp->m_matSimplexTable;
    mySimplex.m_matBiAi = temp->m_matBiAi;
    mySimplex.m_matVar = temp->m_matVar;
    for(i=0;i<nRow;i++)
    {
        for(j=0;j<nCol;j++)
        {
            printf("%lf ", mySimplex.m_matSimplexTable.GetElement(i,j));
        }
        printf("₩n");
    }
    printf("₩n₩n₩n");

    obj_val = mySimplex.m_matSimplexTable.GetElement(nRow-1,nCol-1);
```

는데 결국 메트릭스가 모든 조건을 만족하면 이 매트릭스로부터 우리가 알고자 하는 것, Direction을 모든 조건을 만족시킨 매트릭스로부터 뽑아내면 된다.

우선 m)nPivotCol은 무엇이냐면 FindPivotColumn함수에서 정의되는 값이다. 이것은 처음에 -1로 지정이 되어 있는데 simplextable의 인위목적함수 행의 값들 중에서 음수를 찾고 음수중에서 최소인 열의 열값 int값을 저장하는 클래스 내부변수이다. 결국 우리가 결론적으로 원하는 메트릭스에서는 인위목적함수 행이 모두 양수이므로 m_npivotcol은 아무것도 받아들이지 못한채 -1이 저장되어 있는 것이다. 마찬가지로 m_nPivotrow도 마찬가지다 이런식으로 조건을 만들어 주어서 findpivotcolumn과 findpivotrow함수를 실행시키는 이유는 나중에 모든 함수가 조건을 만족했을 시에 pivot함수를 두번 실행시키지 않기 위함이다. 어쨌든 pivot함수까지 진행하면 인위목적함수 값이 0이 아니면 0이 될때까지 findpivotcolumn과 findpivotrow, pivot함수를 무한 돌린다. 그래서 0이 되었다고 하면 이제 인위목적함수계수가 모두 양수인지를 확인하는데 이때 만족을 못하면 이것은 예초에 잘못된 simplextable이었으므로 roll-back기능을 이용하여 다른 simplextable을 꺼내어 다시 findpivotcolumn과 findpivotrow, pivot함수를 돌린다. 마찬가지로 이것을 만족시킨 table은 마지막 관문인 US가 0인지 확인하는 과정이 남는데 이것마저 통과하면 table은 table에서 뽑을수 있는 정보인 direction을 넘겨주고 함수가 끝나게 된다. 언급을 안했는데 roll-back 을 하기 위해 table을 저장시키는 부분은 findpivotcolumn과 findpivotrow 함수 안에 포함되어 있다. 아래

```
void Simplex::FindPivotColumn() //최소인 음의 값을 가지는 열 선택
{
    int i = 0; //for 문을 위한 index 초기화

    double val = 0.0; //목적 함수의 계수 비교시 초기 설정 값
    m_nPivotCol=-1;

    //Simplex Table에서 목적 함수의 계수가 저장된 열(Row) 값 입력
    int nRow = m_matSimplexTable.GetNumOfRow();
    int nCol = m_matSimplexTable.GetNumOfColumn();

    if(m_nPhase==1)
    {
        for (i=0; i < nCol-1; i++)
        {
            if(m_matSimplexTable.GetElement(nRow - 1, i) < val)
            {
                val = m_matSimplexTable.GetElement(nRow - 1,i); //비교하여 최소인 음수 값을 저장
                m_nPivotCol = i; //최소일 때의 index를 저장
            }
        }
    }
    else
    {
        for (i=0; i < nCol-1; i++)
        {

            if(m_matSimplexTable.GetElement(nRow - 1, i) < val)
            {
                val = m_matSimplexTable.GetElement(nRow - 1,i); //비교하여 최소인 음수 값을 저장
                m_nPivotCol = i; //최소일 때의 index를 저장
            }
        }
    }

    for (i=0; i<nCol-1; i++)
    {
        if ((fabs(m_matSimplexTable.GetElement(nRow-1,m_nPivotCol) -
            m_matSimplexTable.GetElement(nRow-1,i)) < 10e-6) && (i != m_nPivotCol))
        {
            Simplex* temp = new Simplex(*this);
            temp->m_matSimplexTable.PrintElementConsole();
            cout<<m_nPivotCol<<endl;
            temp->m_nPivotCol = i;
            temp->FindPivotRow();
            m_vSimplexStack.push_back(temp);
        }
    }
}
```

의 코드부분이 바로 그부분이다.

4. 탐색방향을 이용한 최소점 탐색

이제 매트릭스를 pivot 한 결과를 바탕으로 강하함수를 구성한다. 강하함수는 R 과 V 의 두 파트로 이루어진다. R 은 Lagrange multiplier 들의 합과 임의 설계 변수의 크기 비교를 통해 구성되며, V 는 제약 조건 함수 값 중 가장 큰 것을 선택하여 결정한다. 이 부분은 PeneltyFunction 함수에서 구현하게 된다. 필요한 데이터를 기존 매트릭스에서 정확하게 추출하여 대입하는 작업이 중요하며, 이번 프로그램에서는 pivot 이 완료된 매트릭스에서 필요한 데이터를 뽑아내어 새로운 배열에 저장시킨 후에 그 배열에 있는 데이터를 이용하도록 했다. V 는 제약조건을 모두 만족시킬 경우 0 이 리턴되므로 최종적으로 목적함수의 최소값을 구할 수 있게 된다. R 와 V 의 구성이 완료되었으면 이 둘을 곱하여 강하함수를 완성한다. 다시 이를 목적함수에 더함으로서 PeneltyFunction 의 구성이 완성된다.

```
V = 0;
if(V < temp1)
{
    V = temp1;
}

if(V < temp2)
{
    V = temp2;
}

ret = m_ObjFn(x) + R*V;
return ret;
```

α Penelty Function 의 구성이 완료되었으면 FindNextMinSection 으로 넘어간다. 이 함수는 제약조건과 목적함수를 바탕으로 현재 위치에서 다음 최적 후보점을 찾는 함수이다. 앞서 탐색 방향과 시작점이 지정되었으므로 이는 1 차원 함수가 된다. 이번 프로젝트에서는 1 차원 황금분할법을 사용하였으며, 따라서 1 차 프로젝트에서 작성한 GoldenSectionSearch 함수와 FindSection 함수를 이번 프로젝트에 맞도록 재구성하였다. 단, 이번 프로젝트에서는 이 두 함수를 합하여 하나의 함수처럼 구성했다. 시작점에서부터 1.618 배 씩 거리를 늘여가며 최소점을 탐색하도록 했고, 함수의 최소값을 갖게 하는 alpha 를 이용하여 다음 탐색 시작점을 구성했다. 최소점이 라 하면 다음 단계의 탐색 시작점은 아래와 같은 식으로 구성된다.

$$x_{n+1} = x_n + \alpha d_x$$

x_n d_x α위 식에서 은 이전 단계의 탐색 시작점, 는 이전 단계의 탐색 방향, 는 Penelty Fuction 을 최소가 되게 하는 점이다.

```
double SQP::find_section(double *direction_)
{
    int j=0;
    int i=0;
    delta = 0.05;
    double norm;
    norm = sqrt(direction_[0]*direction_[0] + direction_[1]*direction_[1]);
    direction_[0] = direction_[0]/norm;
    direction_[1] = direction_[1]/norm;
    for( i=0; i<NumOfVariable; i++)
```

FindSection 함수를 자세히 살펴보면 SQP 클래스 내부에서 정의 되어 있으며, 1 차 프로젝트와는 달리 탐색 방향을 변수로 받아오도록 설정되어 있다는 것을 알 수 있다. 클래스 내부에서 함수를 정의한 이유는 Penelty Function 을 복잡하게 static 함수로 설정할 필요가 없기 때문이며, 탐색

방향을 변수로 받아오도록 한 이유는 시작점을 저장한 변수의 경우 전역변수로 설정되어 FindSection 에서 직접 호출이 가능하지만, 탐색 방향의 경우 FindNextMinPoint 에서의 연산 결과를 바탕으로 정의되어야 하기 때문이다.
FindSection 함수의 최종 리턴 값은 함수의 최솟값이 된다.

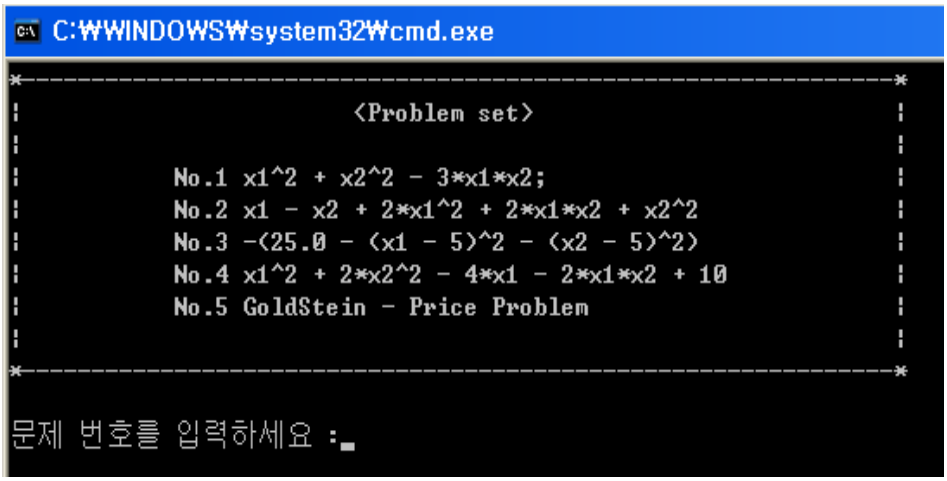
```
    for(i=0;i<n;i++){a[1][i]=Section_x[0][i]+0.618*(Section_x[2][i]-

    lol=(PenaltyFunction(a[1])-PenaltyFunction(a[0]))*(PenaltyFuncti
}
for(i=0;i<n;i++){x_min[i]=a[0][i];}

f_min= PenaltyFunction(a[0]);

return f_min;
```

여기까지가 전체 함수의 한 loop 이다. SQP 클래스 내부의 solve 함수는 이 loop 을 반복적으로 실행하여 종료 조건을 만족하도록 한다.

```
void SQP::Solve()
{

    while(1)
    {

        BuildSimplexTable(); //완성
        FindSearchDirection();
        FindNextMinPoint();

        if(CheckEndCondition() == true)
        {
            break;
        }
```

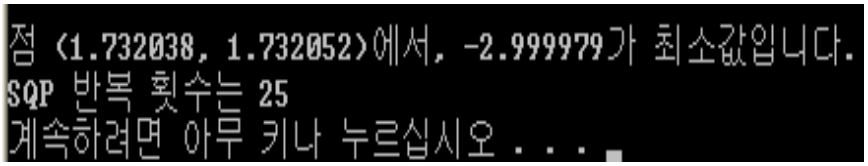
5. 프로그램 실행 결과

이번 프로그램에서 가장 문제가 되었던 부분은 Simplex 클래스의 roll-back 구현이었다. 예외 사항이 꽤 많아 이를 처리하는데 가장 애를 많이 먹었다. 그러나 며칠의 노력 끝에 결국 완벽하게 구현하였으며, 그 결과 1 번 - 5 번 문제 모두 완벽한 답을 얻어낼 수 있었다. 특히 5 번의 경우 Local minimum 과 Global minimum 이 존재하여 시작점에 따라서 최소점이 달리 구해지는 것을 알 수 있었다.



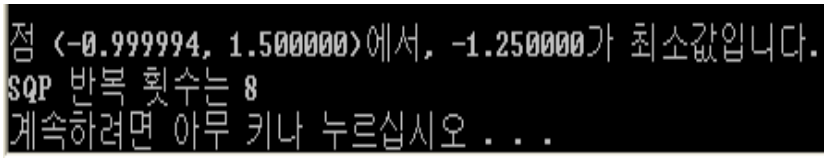
(1) 1 번 문제

1 번 문제의 경우 점(1.732,1.732)에서 최소값 -3 을 가짐을 확인할 수 있었다. 조교님께서 올려주신 초기점 중 안장점인 (0,0)과 (-2,-2)이외에서는 모두 결과값이 나옴을 확인할 수 있었다.



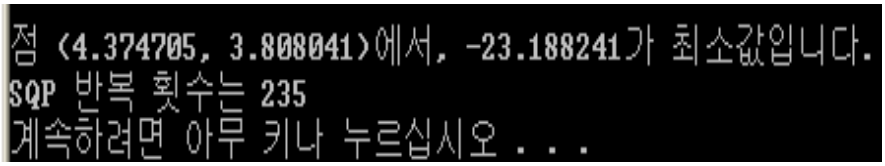
(2) 2 번 문제

2 번 문제의 경우 점(-1,1.5)에서 최소값 -1.25 를 가짐을 확인할 수 있었다. 테스트 한 모든 점에서 실행됨을 확인하였다.



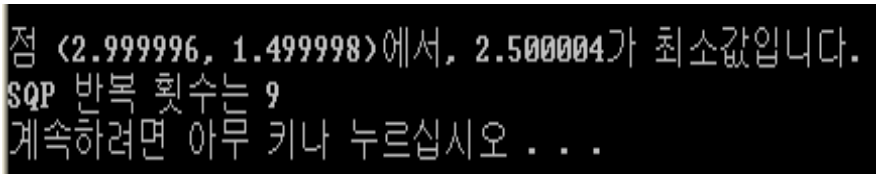
(3) 3 번 문제

3 번 문제의 경우 점(4.374,3.808)에서 최소값 -23.188 을 가짐을 확인할 수 있었다. 테스트 한 모든 점에서 실행됨을 확인하였다.



(4) 4 번 문제

4 번 문제의 경우 점(3,1.5)에서 최소값 2.5 를 가짐을 확인할 수 있었다. 테스트 한 모든 점에서 실행됨을 확인하였다.



(5) 5 번 문제

5 번 문제의 경우 초기점에 따라서 최소값이 달리 출력되는 것을 확인할 수 있었다. (1,1)에서 실행한 결과 점(1.2,0.8)에서 최소값 840 을 가지며, 점(-1,1)에서 실행한 결과 점(0,-1)에서 최소값 3 을 가짐을 확인하였다. 또한, 점 (-0.5,-0.3)에서 실행한 결과 점(-0.6,-0.4)에서 최소값 30 을, 점(2,0.5)에서 실행한 결과 점(1.8,0.2)에서 최소값 84 를 가짐을 확인할 수 있었다. 그러나 어떤 초기점에서는 실행시간이 무척 길어져 runtime error 가 발생하기도 하였는데, 이에 대한 이유는 발견하지 못했다. 이는 점(2,0)에서 실행하였을 때 발생하였다.

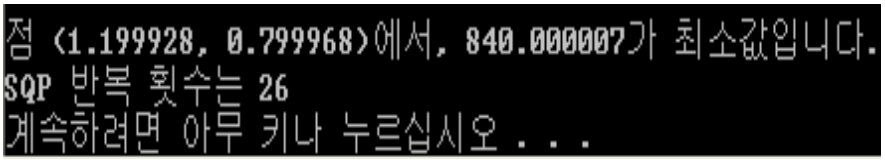


그림 1 5번문제 실행 결과(1,1)

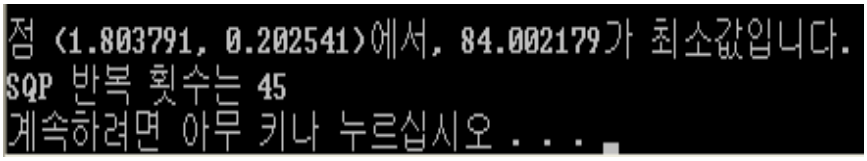


그림 2 5번문제 실행 결과(2,0.5)

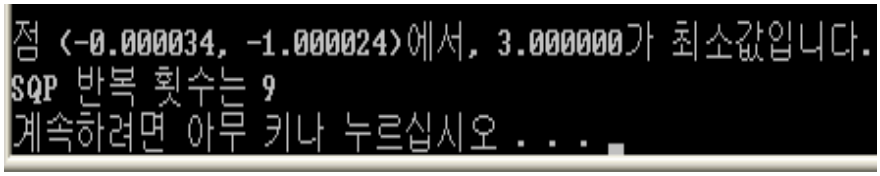


그림 3 5번문제 실행 결과(-1,-1)

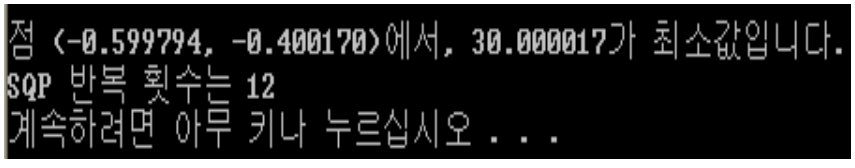


그림 4 5번문제 실행 결과(-0.5,-0.2)

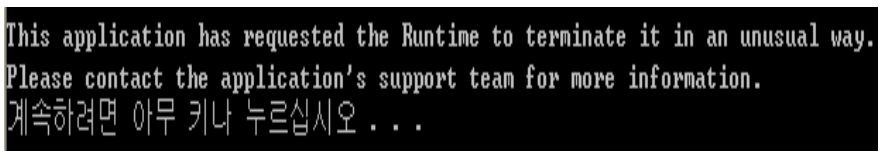


그림 5 5번문제 실행 결과(-2,-2)

6. 후기

- 김재현

-이번 프로젝트는 바로 이전에 수행했던 프로젝트인 Simplex Method와 1차 프로젝트였던 황금분할법을 이용하여 SQP_Method를 구성하는 것이다. 프로젝트를 수행하기 전에는 Simplex 방법과 황금분할법을 반복해서 구하는 것으로 대충 생각했을 때 이번엔 좀 쉬우려니 했는데 이게웬일인지 너무도 복잡하고 어려웠다. 특히나 Simplex 방법에서 Roll_Back 부분이 제대로 구현되지 않은 상태에서 SQP방법을 짜려고 하니 Simplex방법으로 돌아가 Roll-Back 부분을 다시 구현하는 일이 너무 힘들었다. 사실 그게 가장 힘들었던 부분이었는데 Roll_Back부분이 구현이 되었다 하더라도 그 뒤가 첩첩산중이었다. 1번 예제는 돌아가는데 2번이후가 잘 안돌아가는 것은 아직 정확한 원인을 찾지 못했다.

이번프로젝트를 하면서 단순히 프로그램을 짜기위한 지식이 아닌 전반적인 알고리즘을 체계적으로 잘 이해하고 있어야 겠다는 생각이 강하게 들었다. 제대로 이론을 숙지하지 못하니까 프로그램을 짜더라도 조교님이 공지한 ppt에만 계속 의존하게 되고 프로그램이 까딱하면 삐둘어지게 되는 듯 하다.

이번에도 우리 팀원인 배상준 군이 너무 열심히 해주었는데 같은 팀원으로서 내가 너무 도움이 못된 것 같아 미안하고, 항상 질문하러 가면 성심성의껏 도움주는 조교님께 감사하단 말을 남긴다.

- 배상준

-4 차 프로젝트는 그 어느 프로젝트보다도 힘들었다. 3 차 때 roll-back 기능을 정확하게 구현해 놓지 않았던 것이 화를 자초했던 것 같다. 며칠간 밤을 새면서 roll-back 을 구현하려고 노력했으나, 이론에 대한 이해가 정확하지 않아 결국 성공하지는 못했다. 그러나 그 나머지 부분 - Simplex Table 의 구성, 황금분할법을 통한 최적 후보점의 선정 - 에서는 만족할만한 성과를 거뒀다고 생각한다.

이번 프로그래밍 과제에서 얻은 문법 지식이라면 복사 생성자와 소멸자에 대한 개념이다. 이들 역시도 한 번도 사용해보지 않았던 터라 많이 고민했었는데, 조교님께 찾아가서 많이 여쭈어본 결과 이에 대해서 이해할 수 있었다. 그러나 아직 알고리즘이 복잡해지면 그를 다루는 방법이 능숙하지 못하여 이번 프로젝트와 같은 고 난이도의 알고리즘을 짜는 일은 많은 어려움을 느낀다. 이는 프로그래밍을 하면서 점점 나아질 것이라고 믿는다.

3 차까지 잘 해내왔는데 4 차에서 갑자기 너무 높은 벽을 만난 것 같아 당황스럽다. 이번 프로젝트는 비록 미완성으로 제출하지만, 중간고사를 마무리하고 난 후에 이를 꼭 완성시켜 볼 것이다.