[전산선박설계] PA#5

< Determination of Principal particulars of Ship With SQP>

과목명 : 전산선박설계

담당 : 이규열 교수님

제출일 : 11월 8일

공과대학 조선해양공학과

2007-11904, 박민선

2007-11954, 이지혜

Contents

0. SQP 수정본

지난 SQP 제출본이 임의의 모든 초기점에 대해서 값을 찾지 못하는 불안정함을 보였기에 Golden section Search method(GSSM) class와 Simplex class를 수정하여 문제를 해결하였다. 또한 선박의 주요치수를 결정하는 이번 프로젝트는 이전의 프로젝트와 달리 변수의 개수가 2개를 초과하므로 GSSM class를 다변수 문제에 대해서도 해결할 수 있도록 수정하였다.

```
bool CGSSM::FindSection()
{
	Matrix X[3];
	double Y[3];

	double stop_tol = EPS_GSSM;
	bool check;

	while(1)
	{
		int n=2;

		X[0] = X_initial;
		X[1] = X[0] + X_direction*0.1;
		X[2] = X[1] + X_direction*1.618*0.1;

		int limit_num = 100;

		while(1)
		{
			Y[0] = ObjectFunc(X[0]);
			Y[1] = ObjectFunc(X[1]);
			Y[2] = ObjectFunc(X[2]);

			if((Y[1]<Y[0]) && (Y[1]<Y[2]))
			{	check = true;
				break;			}

			if(n > limit_num)
			{	check = false;
				break;			}

			X[0] = X[1];
			X[1] = X[2];
			X[2] = X[2] + X_direction*pow(1.618, n)*0.1;

			n++;
		}
```

```
                if(check)
                        break;
                else
                {
                        double dir = 0;
                        for(int i=0; i<numofvari; i++)
                        {
                            dir += pow(pow(X_direction.GetElement(i, 0), 2),0.5);
                        }
                        if(dir < stop_tol)
                                break;
                }
                X_direction = X_direction * 0.5;
        }

        X_lower = X[0];
        X_middle = X[1];
        X_upper = X[2];

        return check;
}
```

우선, GSSM에서 현재 점과 이전의 점, 다음 점의 각각의 함수 값을 비교하여 최소값이 존재하는 구간을 찾는데 이 때 구간탐색이 세 점 사이의 거리의 크기에 크게 의존하고 있다는 문제점이 발견되었다. 적절하지 않은 값으로 다음점을 찾게 되면 원하는 구간을 얻지 못하고 무한루프를 돌게 되는데, 만약 루프를 도는 횟수가 limit_num, 즉- 100번을 넘게 되면 While문을 빠져나와 탐색거리를 반으로 줄인 값으로 다시 구간탐색을 수행할 수 있도록 수정하였다. 또한 루프를 계속돌았음에도 불구하고 탐색구간을 계속해서 찾지 못할 경우, 무한루프를 방지하기 위해 탐색방향의 크기가 기준값 이하로 가면 while문을 빠져나오도록 하였다. 이를 위해 함수를 bool형태로 바꾸었다.

```
#define EPS_SIMPLEX1 1.0e-6
#define EPS_SIMPLEX2 1.0e-12
                ...

void CSimplex::FindPivotColumn()
{               ...

        for(int i=0; i<m_nnumofcol-1; i++)
        {
        if(fabs(m_matSimplexTable.GetElement(limitrow, i)) < EPS_SIMPLEX2)
                m_matSimplexTable.SetElement(limitrow, i, 0);
        }
                ...
}

void CSimplex::FindPivotRow()
{               ...

        for(int i=0; i<limitrow; i++)
        {
                for(int j=0; j<m_nnumofcol; j++)
                {
                if(fabs(m_matSimplexTable.GetElement(i, j)) < EPS_SIMPLEX2)
                        m_matSimplexTable.SetElement(i, j, 0);
                }
        }
                ...
}
```

Simplex의 1차 미분 값을 구하는 계산과정에서 아주 작은 숫자가 나타나 수치오류 가발생하기 시작하여 SQP가 불안정지게 되었다. Pivot row, pivot column, checkendcondition의 각 단계에서 계산된 값들 중 아주 작은 값은 0으로 고쳐주는 과정을 거쳐 수치오류를 잡을 수 있었는데, 어떤 값을 기준으로하여 값을 0으로 두어야 할지 불분명하였기 때문에 장시간 동안 수차례의 시도 끝에 적정값을 찾는데에 성공하였다.

예제 1번에 대해서 확인해보아야할 점 (0.1, 0.1) , (1, 1) , (0.6, 0.2) 점과 이 외의 (0,0) (1.5, 1.5) (10,1) (5,1) (-2,-2) (1.5,1) (0.3,0.2) 의 점에서도 결과값이 나옴을 확인하였다.

SQP 출력화면

I. Introduction

About Determination of principal particulars of Ship

지금까지 구현했던 다양한 최적화 기법들을 이용하여 실제 선박의 주요 치수(길이, 폭, 깊이, 방형계수)를 결정하는 과제를 수행해보기로 한다.

선주가 요구하는 화물창 용적의 크기, 배 자체가 기본적으로 가져야 하는 부력& 중량의 평형 조건, 국제 규약으로서 주어져 있는 최소 건현 조건 등이 주요치수 변수들로 수식화되어 제약조건으로 주어지며, 이러한 제약 조건을 만족시키는 무수한 해들 가운데 건조비용를 최소화시키는 혹은 운항비용을 최소화시키는 주요치수를 결정짓는 하나의 해를 구하게 된다. 즉- 선박의 주요 치수 결정 문제를 미지수 4개, 2개의 등호제약조건, 1개의 부등호제약조건을 가지는 최적화 문제로 idealize 할 수 있다. 이 과정에서 사용되어지는 여러 값들은 과거에 지어졌던 우수한 실적선들의 값을 이용하여 구한다.

II. Progamming

Function (1) SetRequirement

Main문에서 선주의 요구 조건 세 가지 즉, 재화중량, 화물창 용적, 선속, 그리고 통과하는 운하에 따른 정해진 최대 흘수를 클래스 안으로 받아와서 이를 앞으로 선박 주요치수 계산에 사용하게 된다.

```
void CShip::SetRequirement(double _fDWT_s, double _fCCrequirement, double _fT_d,
double _fVs)
{
        m_fDWT_s = _fDWT_s;
        m_fCCrequirement = _fCCrequirement;
        m_fT_d = _fT_d;
        m_fVs = _fVs;
}
```

Function (2) CalculateParentShipData

기준선의 정보를 이용해 주요치수를 결정하는데 필요한 계수들을 계산한다.

선각중량계수, 의장부중량계수, 기관부중량계수 등을 계산하고 이 값들은 각각 선각, 의장부, 기관부의 중량을 계산하는데에 쓰이게 된다.

```
void CShip::CalculateParentShipData(CShip BaseShip) //
기준선정보로부터관련계수를계산하는함수
{

        // 흘수
        m_fT_s = BaseShip.m_fT_s / BaseShip.m_fT_d * m_fT_d;

        // 기준선정보
        double Lbp_base = BaseShip.m_fLbp;
        double B_base = BaseShip.m_fB;
        double D_base = BaseShip.m_fD;
        double T_s_base = BaseShip.m_fT_s;
        double DWT_s_base = BaseShip.m_fDWT_s;
        double LWT_base = BaseShip.m_fLWT;
        double Ws_base = BaseShip.m_fWs;
        double Wo_base = BaseShip.m_fWo;
        double Wm_base = BaseShip.m_fWm;
        double Cb_s_base = BaseShip.m_fCb_s;
        double Vs_base = BaseShip.m_fVs;
        double CC_base = BaseShip.m_fCC;
        double Fb_base = BaseShip.m_fD - BaseShip.m_fT_s;

        m_fDensity = 1.025;

        // Appendage Factor(1 + alpha)
        m_fAppendageFactor = (DWT_s_base + LWT_base) /
(Lbp_base*B_base*T_s_base*m_fDensity*Cb_s_base) - 1.0;
        // 선각중량계수
        m_fCs = Ws_base / (pow(Lbp_base, 1.6) * (B_base + D_base));     //0.0414;
        // 의장부중량계수
        m_fCo = Wo_base / (Lbp_base * B_base);  //0.1493
        // 기관부중량계수
        m_fCpower = Wm_base / (pow(Lbp_base*B_base*T_s_base*Cb_s_base, 2./3.) *
pow(Vs_base, 3));
        // 화물창용적계수
        m_fCch = CC_base / (Lbp_base*B_base*D_base);
        // 건현계수
        m_fCfb = Fb_base/D_base;
        // 건조비추정을위한선각중량관련계수
        m_fCps = 2247;
        // 건조비추정을위한의장부중량관련계수
        m_fCpo = 6341;
        // 건조비추정을위한기관부중량관련계수
        m_fCpm = 14757;
}
```

Function (3) CalculateWS

선박 자체의 무게는 선각중량, 의장부중량, 기관부중량으로 나눌 수 있는데, 그 중 선각의 중량을 계산하는 부분이다.

$$W_S = C_S \; L^{1.6}(B + D)$$

선각의 중량을 길이의 1.6승, 폭과 깊이에 각각 비례한다고 가정하고, 그 비례상수는 CalculateParentShipData 함수에서 기준선으로부터 구한 선각중량계수값을 이용한다.

Function (4) CalculateWO

선박 LWT 중 의장부중량을 계산한다.

$$W_O = C_O \; BL$$

선박에 들어가는 파이프나 전기배선 등을 포함하는 의장부는 선박의 종단면의 넓이에 비례한다고 가정할 수 있으며, 의장부중량계수는 CalculateParentShipData 함수에서 기준선으로부터 구하였다.

Function (5) CalculateWM

선박의 LWT 중 기관부중량을 계산한다.

$$W_M = C_M \; NMCR$$

기관부중량은 엔진이 낼 수 있는 최대마력이 클수록 커진다고 생각할 수 있으므로 위와 같이 가정한다.

해당 엔진이 낼 수 있는 최대마력은 전달마력(DHP)에 비례하고

$$DHP = 1/C_{ad} \; (LBTC_B)^{\frac{2}{3}} V^3 \quad \textit{이므로,}$$

$$W_M = C_{power} \; (LBTC_B)^{\frac{2}{3}} V^3$$

로 나타낼 수 있다.

Function (6) CalculateLWT

위에서 구한 선각중량, 의장부 중량, 기관부 중량을 모두 더하여 선박의 경화 중량을 아예 계산하는 함수를 만들었다.

```
double CShip::CalculateLWT()
{
        m_fLWT = CalculateWS() + CalculateWO() + CalculateWM();
        return m_fLWT;
}
double CShip::CalculateWS()// 선각중량을계산하는함수
{
        m_fWs = m_fCs*pow(m_fLbp, 1.6)*(m_fB+m_fD);
        return m_fWs;
}
double CShip::CalculateWO()// 의장부중량을계산하는함수
{
        m_fWo = m_fCo*m_fLbp*m_fB;
        return m_fWo;
}
double CShip::CalculateWM()// 기관부중량을계산하는함수
{
        m_fWm = m_fCpower*pow(m_fLbp*m_fB*m_fT_s*m_fCb_s, 2./3.) * pow(m_fVs, 3);
        return m_fWm;
}
```

Function (7) CalculateBuildingCost

건조비의 최소화. 라는 목적함수를 정의하는 함수로 기본적으로 건조비는 선박의 자체 중량에 비례한다고 가정한다. 앞에서 구한 선각중량, 의장부중량, 기관부중량, 그리고 각 중량에 대해서 주어진 건조비 계수를 이용하면 건조비에 대한 함수를 주요치수에 대해서 정의할 수 있다.

BuildingCost =

$$C_{PS}\,C_s\;\; L^{1.6}(B+D) + C_{PO}\,C_o\;\; BL + C_{PM}\,C_{power}\;\; (LBTC_B)^{\frac{2}{3}}V^3$$

```
double CShip::CalculateBuildingCost(Matrix x)// 건조비를계산하는함수
{
        m_fLbp = x.element[0][0];
        m_fB = x.element[1][0];
        m_fD = x.element[2][0];
        m_fCb_s = x.element[3][0];

        return (m_fCps*CalculateWS() + m_fCpo*CalculateWO() +
                m_fCpm*CalculateWM())*1.0e-8;
}
```

Function (8) CalculateCC

화물창 용적을 계산하는 함수로 화물창의 용적은 배의 전체 부피에 비례할 것이라는 생각하에 아래와 같이 가정한다.

$$V_{H\_req} = C_H \;\; LBD$$

```
double CShip::CalculateCC()// 화물창용적을계산하는함수
{
        m_fCC = m_fCch*m_fLbp*m_fB*m_fD;
        return m_fCC;
}
```

Function (9) CalculateFB

건현은 선박의 예비부력으로써, 운동 중 혹은 횡경사가 심하여 갑판에 물이 들어오게 되어 선박의 무게가 커지게 될 경우를 대비하기 위함이다. 깊이가 깊으면 건현도 커지게 될 것이므로 개략적으로 다음과 같이 가정한다.

$$D \geq T + FB$$

$$D \geq T + C_{FB} D$$

더 구체적으로 본다면, 선박의 길이가 길어지면 횡경사되었을 때 갑판에 물이 들어오는 부분도 커지게 될 것이므로 길이와도 관련이 있을 것이다.

```
double CShip::CalculateFB()// 건현을계산하는함수
{
        m_fFB = m_fCfb*m_fD;
        return m_fFB;
}
```

Function (10) BuoyancyDisplacementCondition

선박은 물 위에 떠 있으므로 그 중량만큼 부력을 받게 될 것이라는 자연의 조건을 만족해야한다. 중량은 선주가 요구하는 재화중량과 선박 자체의 중량으로 이루어지는데, LWT는 앞에서 구한 선각중량, 의장부중량, 기관부중량의 합으로 나타낼 수 있다.

여기서 말하는 중량은 중량가속도 g를 곱하지 않은 질량값이다.

$$\text{Buoyancy} = \text{DWT} + \text{LWT}$$

$$= \text{DWT} + C_s L^{1.6}(B + D) + C_o BL + C_{power} (LBTC_B)^{\frac{2}{3}} V^3$$

$$\text{Weight} = LBTC_B \rho_{SW} C_\alpha$$

Buoyancy – weight 로 등호제약조건으로 수식화한다. 제약조건을 만족하면 0이 된다.

```
double CShip::BuoyancyDisplacementCondition(Matrix x)// 부력-
중량평형조건을계산하는함수
{
        m_fLbp = x.element[0][0];
        m_fB = x.element[1][0];
        m_fD = x.element[2][0];
        m_fCb_s = x.element[3][0];

        double Buoyancy;
        double Weight;

        Buoyancy = m_fLbp*m_fB*m_fT_s*m_fCb_s*m_fDensity*(1+m_fAppendageFactor);
        Weight = m_fDWT_s + CalculateLWT();

        return Buoyancy/Weight - 1.0;
}
```

Function (11) CCRequirementCondition

앞에서 구한 화물창 용적이 선주가 요구하는 화물창 용적과 같아지도록 제약조건을 만든다. 화물창 용적이 선주가 요구하는 값을 만족하면 제약조건의 값은 0이 된다.

```
double CShip::CCRequirementCondition(Matrix x)// 화물창요구조건을계산하는함수
{
        m_fLbp = x.element[0][0];
        m_fB = x.element[1][0];
        m_fD = x.element[2][0];
        m_fCb_s = x.element[3][0];

        return CalculateCC()/m_fCCrequirement - 1.0;
}
```

Function (12) FBRequirementCondition

CalculateFB함수에서 가정한 부등식을 제약 조건으로 준다.

```
double CShip::FBRequirementCondition(Matrix x)// 건현요구조건을계산하는함수
{
        m_fLbp = x.element[0][0];
        m_fB = x.element[1][0];
        m_fD = x.element[2][0];
        m_fCb_s = x.element[3][0];

        return (m_fT_s + CalculateFB())/m_fD - 1.0;
}
```

Function (13) ObesityCoefficientCBCondition

배의 뚱뚱하고 날씬한 정도를 나타내는 방형계수 C_B는 값이 작을수록 선형이 날씬해서 물을 잘 가르고 나가므로 저항과 추진 성능이 좋고, 연료 소모량이 줄어들게 된다. 그러나 C_B는 배의 복원성능과도 관련이 있기 때문에 저항성능을 위해 방형계수를 낮추다

보면 복원성능이 떨어지는 배가 된다. 그러므로 적정한 방형계수 값을 가지도록 하는 것이 중요한데, 선박이 적절한 조종성능을 가지도록 하는 방형계수 조건으로 비만계수요구조건을 준다.

$$\frac{C_B}{\frac{L}{B}} \leq 0.15$$

```
double CShip::ObesityCoefficientCBCondition(Matrix x)//
조종성관점에서의비만계수요구조건
{
        m_fLbp = x.element[0][0];
        m_fB = x.element[1][0];
        m_fD = x.element[2][0];
        m_fCb_s = x.element[3][0];

        return m_fCb_s / (m_fLbp / m_fB) /0.15 - 1.0;
}
```

Function (14) WGCBCondition

Watson & Gilfillan이 방형계수 값을 추정하도록 세운 식을 현재 많이 사용하고 있으므로 이 식도 조건으로 넣어준다. F_n는 Froude number.

$$C_B \leq 0.7 + 0.125 * \arctan((23-100F_n)/4)$$

```
double CShip::WGCBCondition(Matrix x)// Watson & Gilfillan에의한Cb추천값
{
        m_fLbp = x.element[0][0];
        m_fB = x.element[1][0];
        m_fD = x.element[2][0];
        m_fCb_s = x.element[3][0];

        double Fn = m_fVs*0.5144 / sqrt(9.81*m_fLbp);

        return m_fCb_s - (0.70 + 0.125*atan((23.0-100.0*Fn)/4.0));
}
```

Function (15) SetDVUpperLowerBoundary

우리가 이번 프로그램의 결과로 얻고자 하는 값인 선박의 길이, 폭, 높이, 그리고 방형계수가 터무니 없는 값이 나오지 않고 적합한 범위를 갖도록 각 변수에 대해서 상하 한계를 설정하는 것도 필요하다. 이렇게 되면 네 개의 변수에 상, 하한의 부등호 제약조건이 추가되기 때문에 총 8개의 부등호 제약조건이 추가되게 된다.

```
void CShip::SetDVUpperLowerBoundary(Matrix x_l, Matrix x_u)
{
        m_fLbp_lower = x_l.element[0][0];
        m_fB_lower = x_l.element[1][0];
        m_fD_lower = x_l.element[2][0];
        m_fCb_s_lower = x_l.element[3][0];
        m_fLbp_upper = x_u.element[0][0];
        m_fB_upper = x_u.element[1][0];
        m_fD_upper = x_u.element[2][0];
        m_fCb_s_upper = x_u.element[3][0];
}
```

Main문에서 각 변수의 상, 하의 한계 값을 갖는 행렬을 받아와서 이 값을 이용하게 된다. Main 문에서 지정해준 한계 값의 행렬은 다음과 같다.

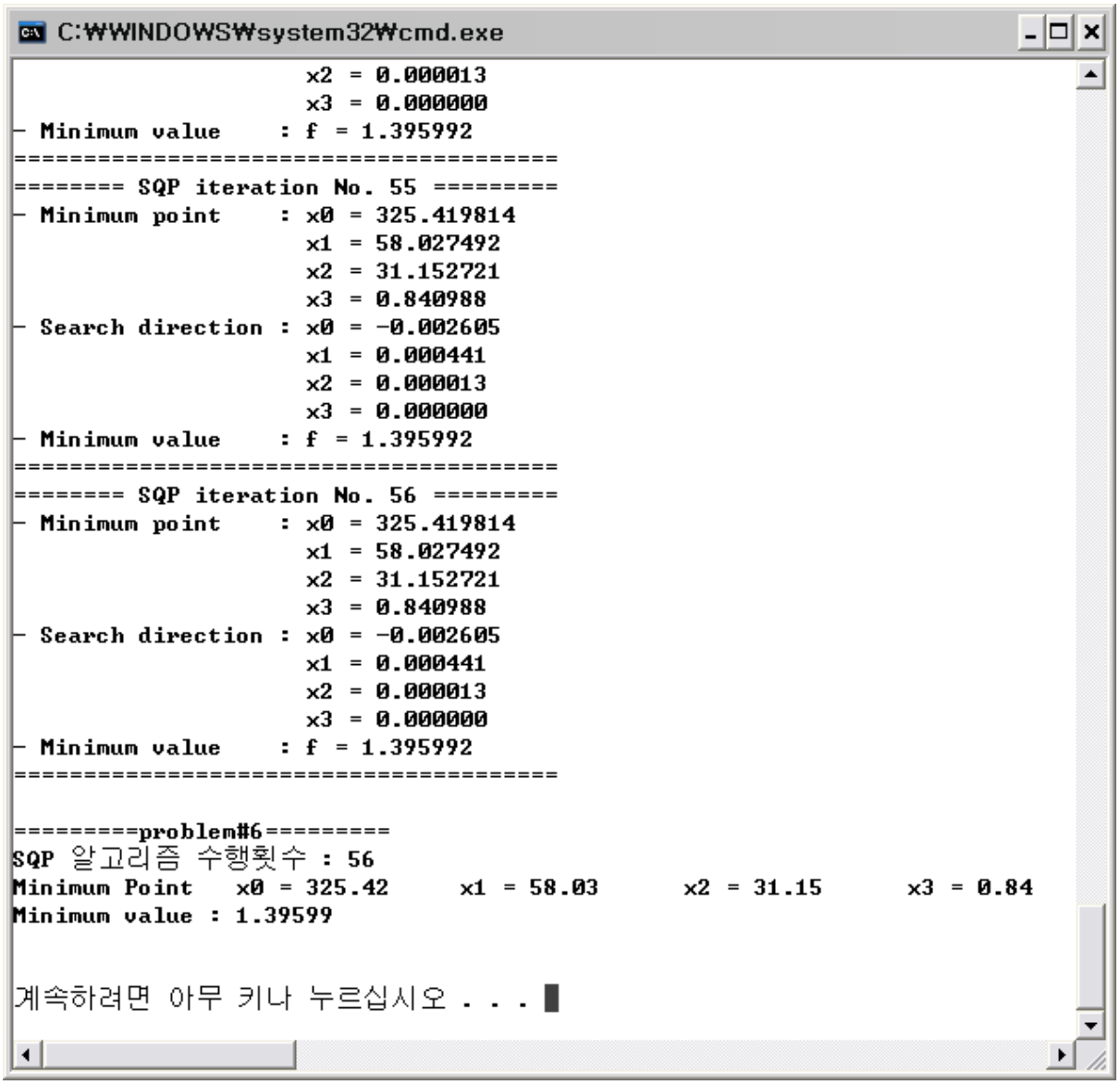
```
Matrix x_l(NumOfVariable, 1);
x_l.SetElement(0, 0, 300);
x_l.SetElement(1, 0, 50);
x_l.SetElement(2, 0, 20);
x_l.SetElement(3, 0, 0.8);

Matrix x_u(NumOfVariable, 1);
x_u.SetElement(0, 0, 350);
x_u.SetElement(1, 0, 70);
x_u.SetElement(2, 0, 40);
x_u.SetElement(3, 0, 0.9);
```

Function (15) DVUpperCondition & DVULowerCondition

SetDVUpperLowerBoundary에서 상하 한계 값의 행렬을 받아왔으므로 이 값을 이용하여 현재 계산 과정에서의 주요 치수와 각 한계 값의 오차를 넘겨주는 함수를 만들어 주었다. 이 상하 한계 값이 새로운 부등호 제약조건이 되어 8개의 부등호 제약조건이 추가되는 셈이다. 이렇게 부등호 제약조건이 많아지면 계산 속도가 느려지고 올바른 답을 얻기까지 수정해야 할 코드가 많아지지만 다양한 경우에 대해서도 합당한 범위 내의 주요 치수를 얻을 수 있을 것이다.

```
double CShip::DVLowerCondition(int n, Matrix x)//
설계변수의하한값에대한조건을계산하는함수
{
	m_fLbp = x.element[0][0];
	m_fB = x.element[1][0];
	m_fD = x.element[2][0];
	m_fCb_s = x.element[3][0];
	double output;
	switch(n)
	{
		case 0:
			output = m_fLbp_lower - m_fLbp;
			break;
		case 1:
			output = m_fB_lower - m_fB;
			break;
		case 2:
			output = m_fD_lower - m_fD;
			break;
		case 3:
			output = m_fCb_s_lower - m_fCb_s;
			break;
	}
	return output;
}
double CShip::DVUpperCondition(int n, Matrix x)//
설계변수의상한값에대한조건을계산하는함수
{
	m_fLbp = x.element[0][0];
	m_fB = x.element[1][0];
	m_fD = x.element[2][0];
	m_fCb_s = x.element[3][0];
	double output;
	switch(n)
	{
	case 0:
		output = m_fLbp - m_fLbp_upper;
		break;
	case 1:
		output = m_fB - m_fB_upper;
		break;
	case 2:
		output = m_fD - m_fD_upper;
		break;
	case 3:
		output = m_fCb_s - m_fCb_s_upper;
		break;
	}
	return output;
}
```



Ⅲ. 후기

지금까지 수학적인 문제의 최소값을 찾기 위해 최적화 방법으 사용한 것과는 달리, 이번에는 숫자가 아닌 실제 사회적, 물리적 제약 조건들 속에서 조건들을 가정하고, 수식화하는 등, 공학문제에 대해 최적해를 얻어내는 연습을 해보았다. 수학문제에 대해서는 답이 잘 나오든 잘 못 나오든 아무런 상관이 없지만 이러한 공학문제는 구한 답이 실제 배를 건조하는데에 핵심적인 데이터이므로 더 신중하게, 더 정확한 해를 찾을 수 있어야 할 것이란 생각이 든다. 물론 더 정확하고 확실한 해를 구하기 위해서 프로그램을 더 섬세하게 짜야한다. 우리 조 또한 임의의 모든 초기점에서 해를 구할 수 있도록 하기 위해서 많은 시간 동안 노력을 계속하였다. 이는 결국 컴퓨터의 계산 상 수치 오차와의 싸움이었다.

1차 미분 값을 계산하는 과정에서 나타난 매우 작은 값들을 보정해주었어야 했는데 제대로 된 최적점을 구하기 위해서 어느 정도를 기준으로 잡고 값들을 보정해 주어야 하는지가 모호했기 때문에 어려움이 많았다. 프로그램 상에서 충분히 작지 않은 값을 0으로 바꿔버리면 심플렉스 계산을 더 이상 할 수 없어지고, 또 0으로 바꾸는 기준을 너무 작은 값으로 해버리면 최적점을 구하는 과정을 지나쳐버리기 때문이었다. 게다가 모든 점에 대해서 최적점을 구할 수 있어야 했기 때문에 그 과정은 더욱 복잡했다.

또한 몇 가지 함수에서는 수렴 범위를 정해놓고, 우리가 지적한 값의 크기가 수렴 범위 안에 들어오면 조건을 만족한다고 판단하게 하는 것이 있었다. 예를 들어 Simplex의 Roll Back 여부라던가, SQP나 Simplex의 계산 종료 조건 등이 그것이다. 그런데 이 때마다 서로 다른 오차범위를 주어주는 것이 임의의 점에서 해를 찾는데 더 도움이 된다는 것을 수많은 시도와 디버깅 끝에 알아낼 수 있었다. 따라서 서로 다른 세 개의 수렴 범위를 정의하여 이를 유연하게, 수치 오차가 적어지는 쪽으로 지정해주었다.

이전까지의 과제는 뭔가 합리적이었다는 느낌이었다. 알고리즘을 바르게 생각하여 미처 생각하지 못한 부분을 보완하고 더 효율적으로 프로그램을 수정하면 항상 좋은 방향으로 나아갔던 것 같다. 하지만 이번 과제는 Simplex가 올바르게 수행됨에도 불구하고, SQP 계산이 제대로 이루어짐에도 불구하고 어떤 점을 초기에 설정하느냐에 따라 답이 나오지 않는 이유도 다르고, 수치 오차도 달랐다. 결국 이는 수많은 디버깅 과정을 통해서 더 나은 코드를 알아내고 조합해 최선의 결과를 얻어야 했다. 고민한 시간도 가장 길고 스트레스도 감당하기 힘들 정도였다. 일주일이 넘도록 컴퓨터 앞에서 끊임없는 디버깅으로 수치오차를 찾아내고, 이를 막아서 임의의 점에 대해 값을 얻는데 한 걸음 더 나아가고 하는 과정 중 결국 금요일 즈음에서는 어느 정도 보완되어 가는 것을 느꼈을 때 너무 기뻤다. 두 명이 서로 컴퓨터를 붙잡고 수치 오차가 발생하는 이유를 하나 찾으면 알려주고 먼저 보완이 되는 사람의 코드를 받아들여 함께 수정함으로써 잘 해결 할 수 있었고, 오랜 시간 공을 들인 결과이기에 더욱 뿌듯한 마음이 든다.