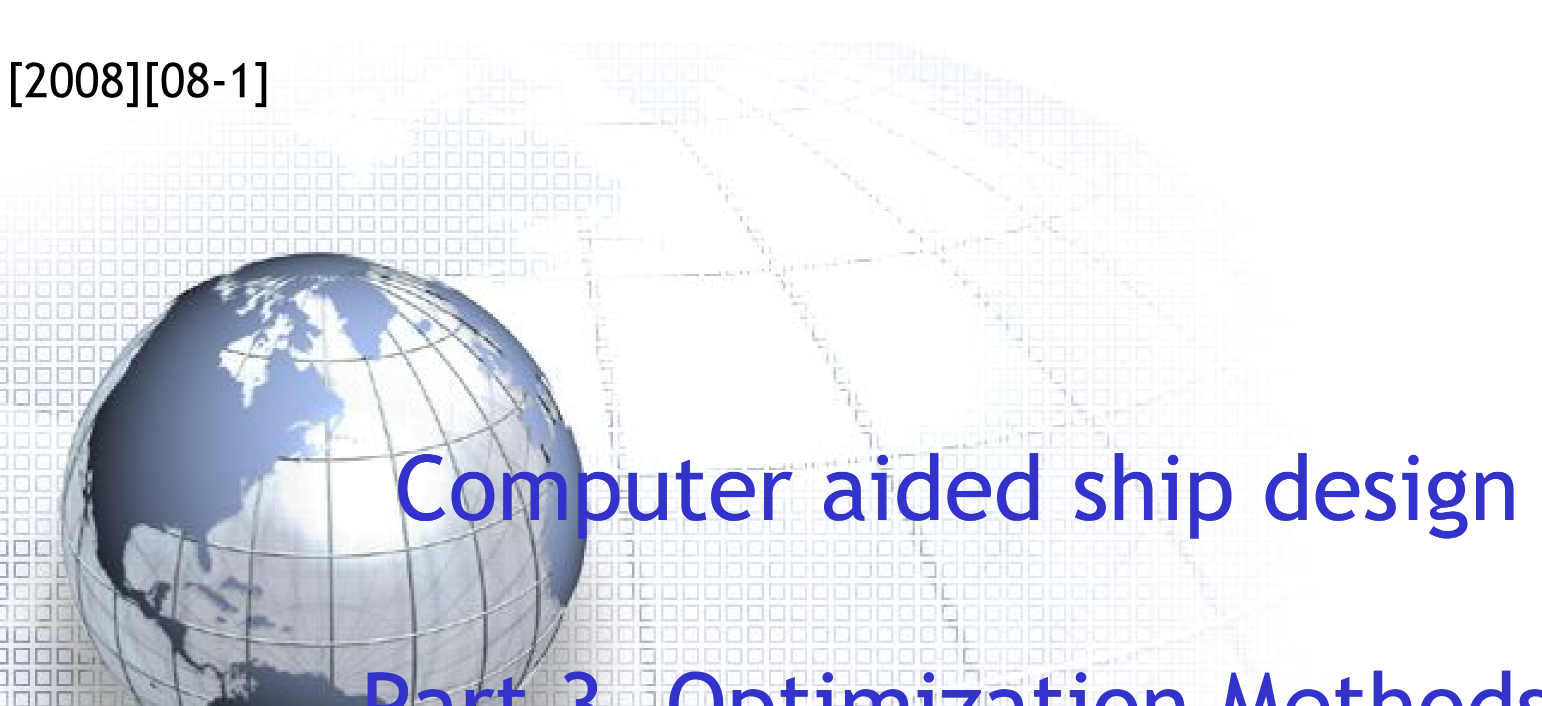
[2008][08-1]



Computer aided ship design

Part 3. Optimization Methods

October 2008

Prof. Kyu-Yeul Lee

Department of Naval Architecture and Ocean Engineering,
Seoul National University of College of Engineering

Advanced
Ship
Design
Automation
Laboratory

2.3 직접 탐사법 (Direct Search Method)

Advanced
Ship
Design
Automation
Laboratory

2.3 직접 탐사법(Direct Search Method)
- Hooke & Jeeves의 직접 탐사법

1. Base Point가 계산 됨
2. Global Pattern Move
3. Local Pattern Search

- 기준점에서의 Local Pattern Search와 최적해 방향으로 가속화하는 Global Pattern Move를 이용해 최적해를 찾는 방법

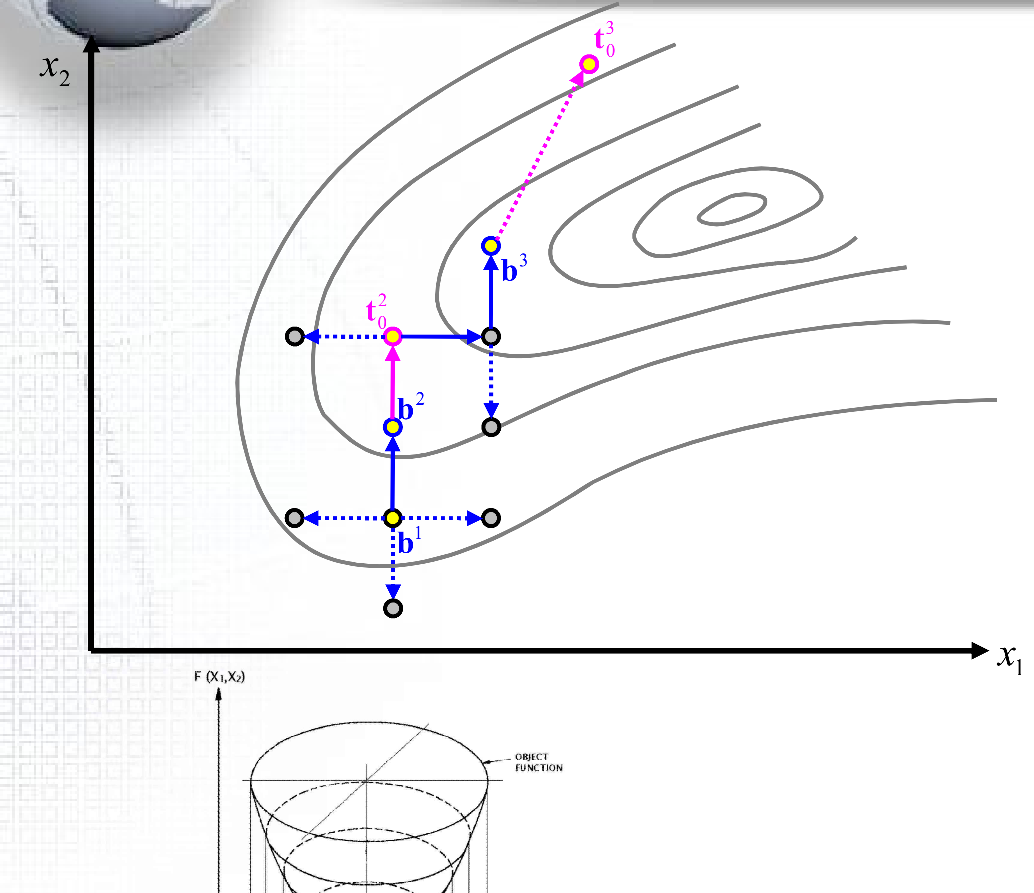
F (X1,X2)
OBJECT FUNCTION
X1
X2
CONTOUR LINE
OPTIMUM (MINIMUM)

x_2
t_0^3
4
7
t_0^5
b^4
$t_0^4 \Rightarrow b^5$
b^3
5
t_0^2
6
b^2
3
b^1
2
Global Pattern Move
기준점
0
x_1
1
Local Pattern Search

2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법 예시 (1/3)

1. Base Point가 계산 됨
2. Global Pattern Move
3. Local Pattern Search



1. 'Local Pattern Search' at point b^1
   - x_1 방향 탐색
     목적 함수 개선이 없음 → x_1 방향 이동 없음
   - x_2 방향 탐색
     목적 함수 개선이 있음 → $+x_2$ 방향 이동
   - 최종 이동 후 base point b^2 정의

2. 'Global Pattern Move' at point b^2
   - b^2에서 b^1을 대칭시켜 임시점 t_0^2를 구함
   - t_0^2에서 목적 함수 값이 b^2보다 개선되었으므로 t_0^2에서 Local Pattern Search 수행

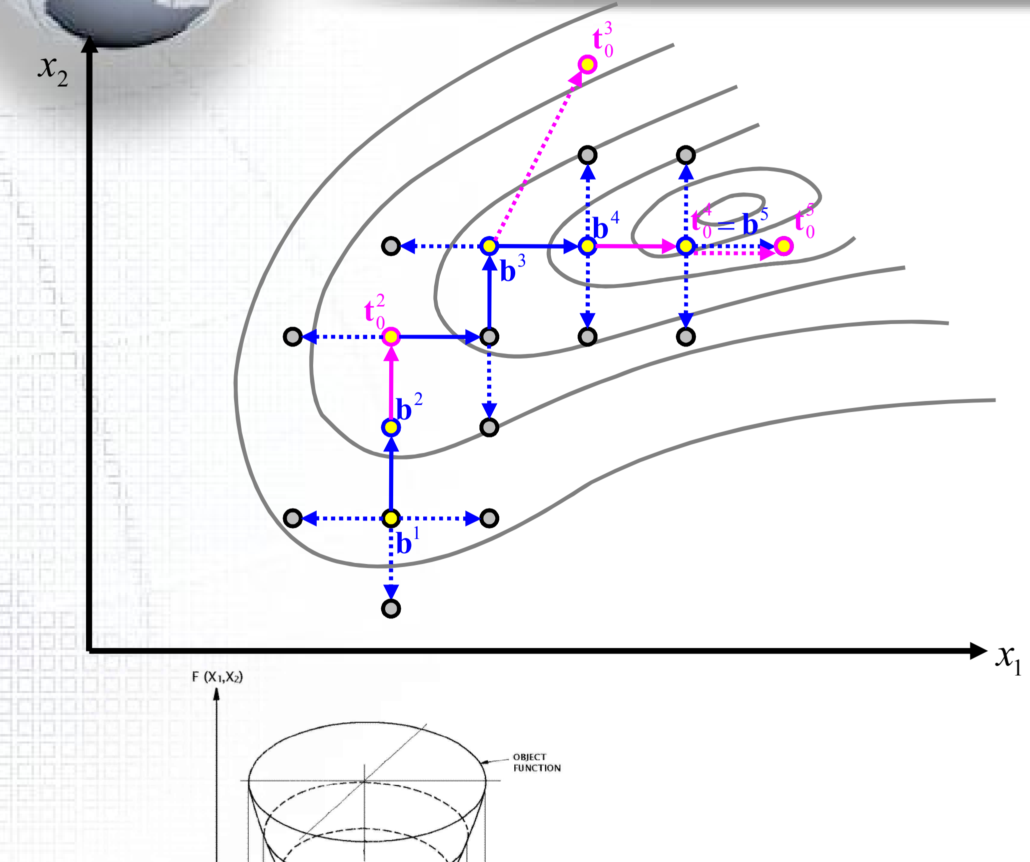
3. 'Local Pattern Search' at point t_0^2
   - x_1 방향 탐색
     목적 함수 개선이 있음 → $+x_1$ 방향 이동
   - x_2 방향 탐색
     목적 함수 개선이 있음 → $+x_2$ 방향 이동
   - 최종 이동 후 base point b^3 정의

4. 'Global Pattern Move' at point b^3
   - b^3에서 t_0^3을 대칭시켜 임시점 t_0^3를 구함
   - t_0^3에서 목적 함수 값이 b^3보다 개선되지 않았으므로 b^3에서 Local Pattern Search 수행

2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법 예시 (2/3)

1. Base Point가 계산 됨
2. Global Pattern Move
3. Local Pattern Search



5. 'Local Pattern Search' at point b^3

- x_1 방향 탐색
  목적 함수 개선이 있음 → $+x_1$ 방향 이동
- x_2 방향 탐색
  목적 함수 개선이 없음 → x_2 방향 이동 없음
- 최종 이동 후 base point b^4 정의

6. 'Global Pattern Move' at point b^4

- b^4에서 b^3을 대칭시켜 임시점 t_0^4를 구함
- t_0^4에서 목적 함수 값이 b^4보다 개선되었으므로 t_0^4에서 Local Pattern Search 수행

7. 'Local Pattern Search' at point t_0^4

- x_1 방향 탐색
  목적 함수 개선이 없음 → x_1 방향 이동 없음
- x_2 방향 탐색
  목적 함수 개선이 없음 → x_2 방향 이동 없음
- 목적 함수의 개선이 없으므로 현재 위치를 base point b^5로 정의

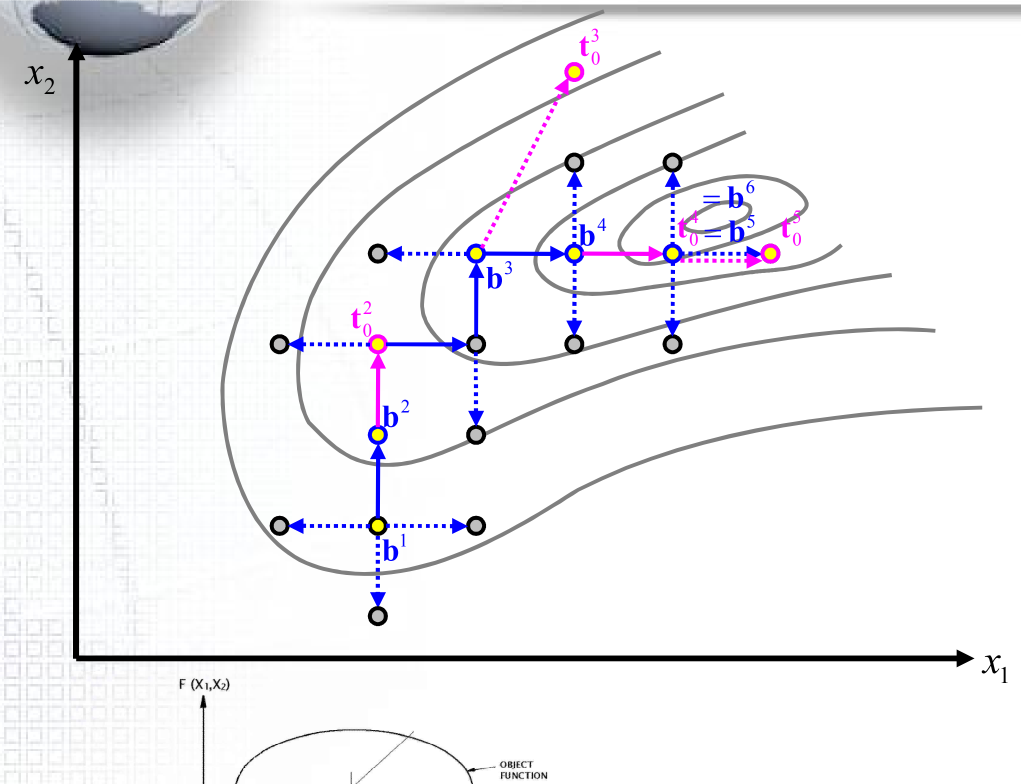
8. 'Global Pattern Move' at point b^5

- b^5에서 b^4을 대칭시켜 임시점 t_0^5를 구함
- t_0^5에서 목적 함수 값이 b^5보다 개선되지 않으므로, b^5에서 Local Pattern Search 수행

2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법 예시 (3/3)

1. Base Point가 계산 됨
2. Global Pattern Move
3. Local Pattern Search



9. 'Local Pattern Search' at point b^5

- x_1 방향 탐색
  목적 함수 개선이 없음 → x_1 방향 이동 없음
- x_2 방향 탐색
  목적 함수 개선이 없음 → x_2 방향 이동 없음
- 목적 함수의 개선이 없으므로 현재 위치를 base point b^6로 정의
- $b^5 = b^6$ 이므로 step size를 ½로 줄이고 Local Pattern Search 수행

F (X1,X2)
OBJECT FUNCTION
X1
X2
CONTOUR LINE
OPTIMUM (MINIMUM)

2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법: Local Pattern Search 규칙 (1/2)

Local Pattern Search의 일반적 규칙

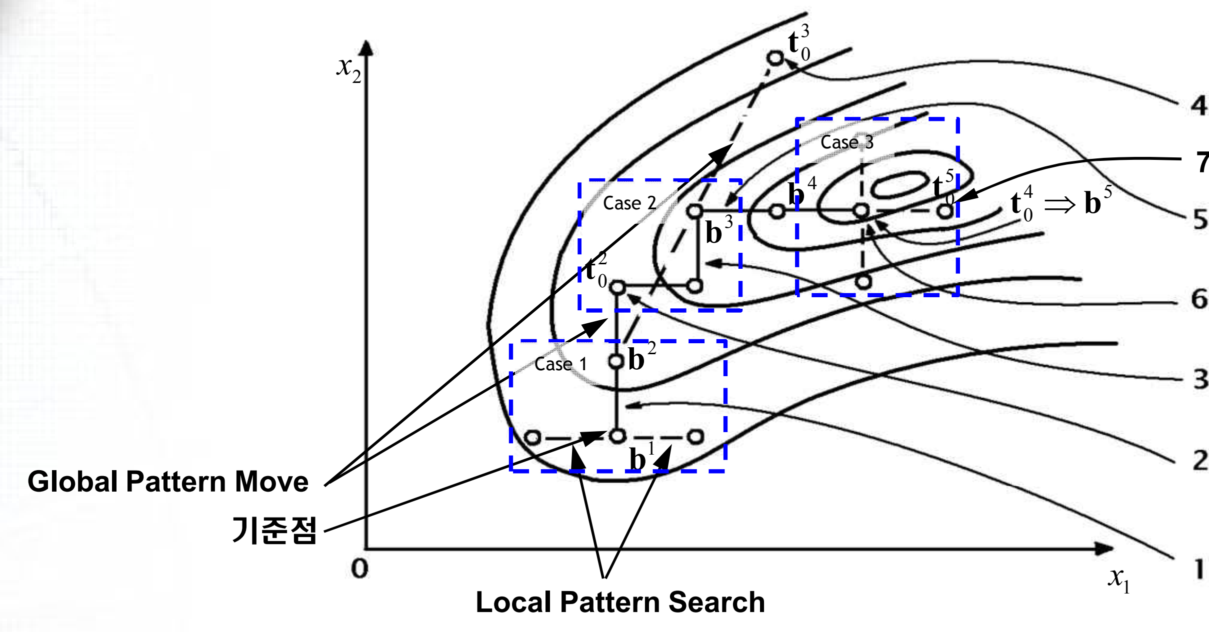
(F: Fail, S: Success)

<table>
<tr><td colspan="3">① x_i 양의 방향 탐색</td></tr>
<tr><td rowspan="2">- x_i 양의 방향으로 Step size만큼 탐색 점을 이동시킨 후 함수값을 비교한다
b^k</td><td>- 이동 후의 점 에서 함수값이 더 크다면</td><td>- 이동 전의 점으로 돌아와 x_i 음의 방향으로 탐색을 한다.
b^k F</td></tr>
<tr><td>- 이동 후의 점 에서 함수값이 더 작다면</td><td>- 이동 후의 점에서 x_{i+1} 방향으로 탐색을 시작한다.
b^k S</td></tr>
<tr><td colspan="3">② x_i 음의 방향 탐색</td></tr>
<tr><td rowspan="2">- x_i 음의 방향으로 Step size만큼 탐색 점을 이동시킨 후 함수값을 비교한다
(x_i 양의 방향으로 탐색이 실패할 경우에만 음의 방향으로 탐색을 실시한다.)
b^k F</td><td>- 이동 후의 점 에서 함수값이 더 크다면</td><td>- 이동 전의 점으로 돌아와 x_{i+1} 방향으로 탐색을 시작 한다.
F b^k F</td></tr>
<tr><td>- 이동 후의 점 에서 함수값이 더 작다면</td><td>- 이동 후의 점에서 x_{i+1} 방향으로 탐색을 시작 한다.
S b^k F</td></tr>
</table>

- i를 1부터 변수의 개수 n까지 증가시키면서 탐색한다.
- n번째 변수까지 탐색을 마치면 그 점을 새로운 Base Point b^{k+1}로 정의한다.

2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법: Local Pattern Search 규칙 (2/2)



* 위 첨자 k는 step 번호를 의미 한다.

- **Local Pattern Search** (F: Fail, S: Success)

<Case 1>

S
b^{k+1}
F — b^k — F

<Case 2>

S
b^{k+1}
b^k — S

<Case 3>

F
F — b^k — F
F

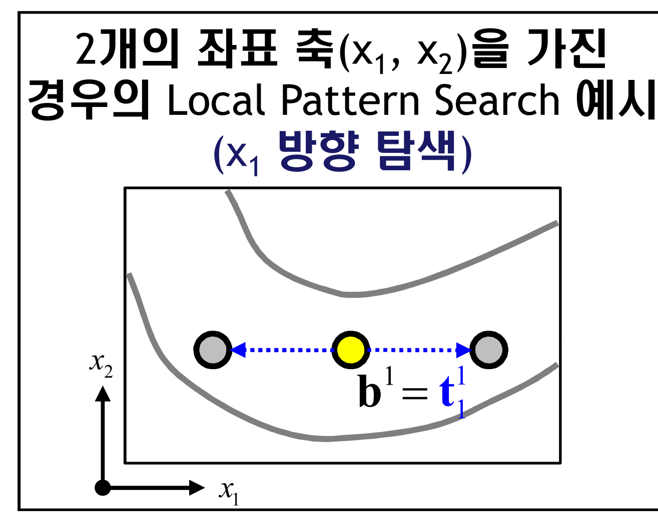
Step/2 ⇒ b^k

2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법의 알고리즘 요약 (1)

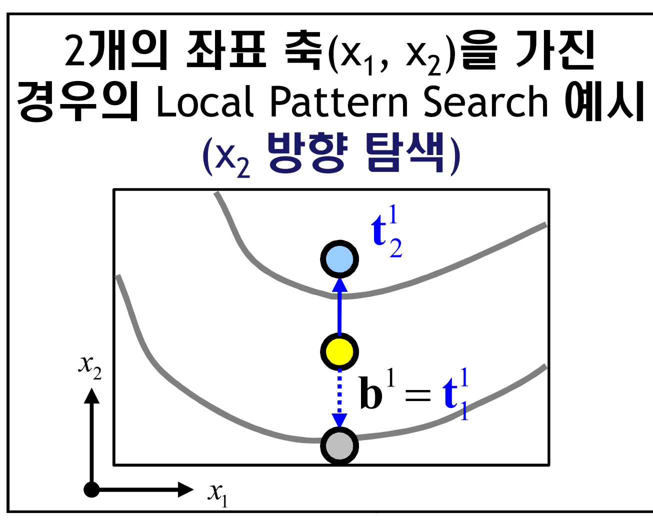
1) Local Pattern Search (총 n개의 좌표 축을 가진 경우에 대하여)

1. 기준점(Base Point) $\mathbf{b}^1$에서의 함수 f의 값을 계산한다.



2. 점 $b^1 \pm \delta_1$ 에서의 함수 f의 값을 계산한다. 여기서 δ_1은 Input Step Size이며 $\delta_1 = [\delta_1, 0, 0, \ldots, 0]^T$로 표현되는 n개의 원소를 가지는 벡터이다. 함수 값의 개선이 있는 점을 t_1^1이라 놓고 이로부터 계속 탐사를 진행한다.

3. 점 $t_1^1 \pm \delta_2$ 에서의 함수 f의 값을 계산한다. 여기서 δ_2는 역시 Input Step Size이며 $\delta_2 = [0, \delta_2, 0, \ldots, 0]^T$ 이다. 그리고 함수 값의 개선이 있는 점을 t_2^1 이라 놓는다.



2.3 직접 탐사법(Direct Search Method)

- Hooke & Jeeves의 직접 탐사법의 알고리즘 요약 (2)

1) Local Pattern Search (**총 n개의 좌표 축을 가진 경우에 대하여**)

4. **나머지 좌표 축에 대해 위와 같은** Local Pattern Search **과정을 수행하고 새로운 기준점**(New Base Point)**을 설정한다. 즉,** Local Pattern Search **과정이 끝나면 새로운 기준점이 선정된다.** (**선정된 새로운 기준점** $b^2 = t_n^1$)

5. **새로운 기준점과 그 직전 기준점의 방향으로** Global Pattern Move**를 수행한다.**

2.3 직접 탐사법(Direct Search Method)
- Hooke & Jeeves의 직접 탐사법의 알고리즘 요약 (3)

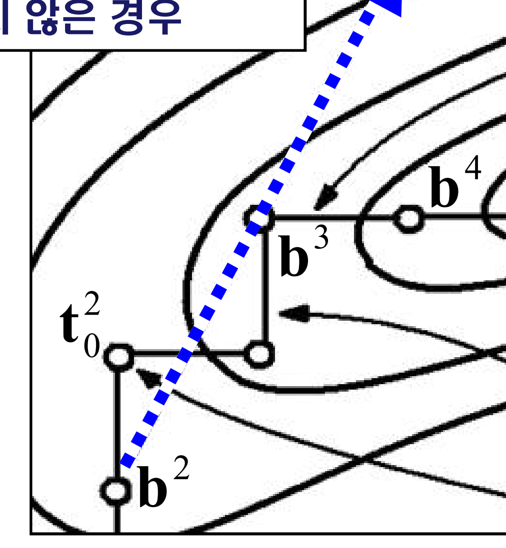
2) Global Pattern Move

1. Local Pattern Search에 의해 얻어진 2개의 기준점을 연결하는 방향을 따라 제일 마지막 기준점에서 이 두 기준점 사이의 거리만큼 떨어져 있는 점을 임시 기준점(Temporary Point)로 설정하고(“Global Pattern Move”), 임시점에서의 함수 f 의 값을 계산한다. Global Pattern Move에 의해 선정된 임시 기준점은 다음과 같다.

$$\mathbf{t}_0^{k+1} = \mathbf{b}^k + 2(\mathbf{b}^{k+1} - \mathbf{b}^k) = 2\mathbf{b}^{k+1} - \mathbf{b}^k$$

2. 만일 이 임시점에서 함수 f 의 값이 개선된다면 이 점에서부터 다시 Local Pattern Search를 수행한다. 함수 f 의 값이 개선되지 않는다면 그 직전의 기준점으로 돌아가 Local Pattern Search를 수행한다.

2개의 좌표 축을 가진 경우의 Global Pattern Move의 예시
임시 기준점에서의 함수 값이 개선되지 않은 경우



3) Closing Conditions

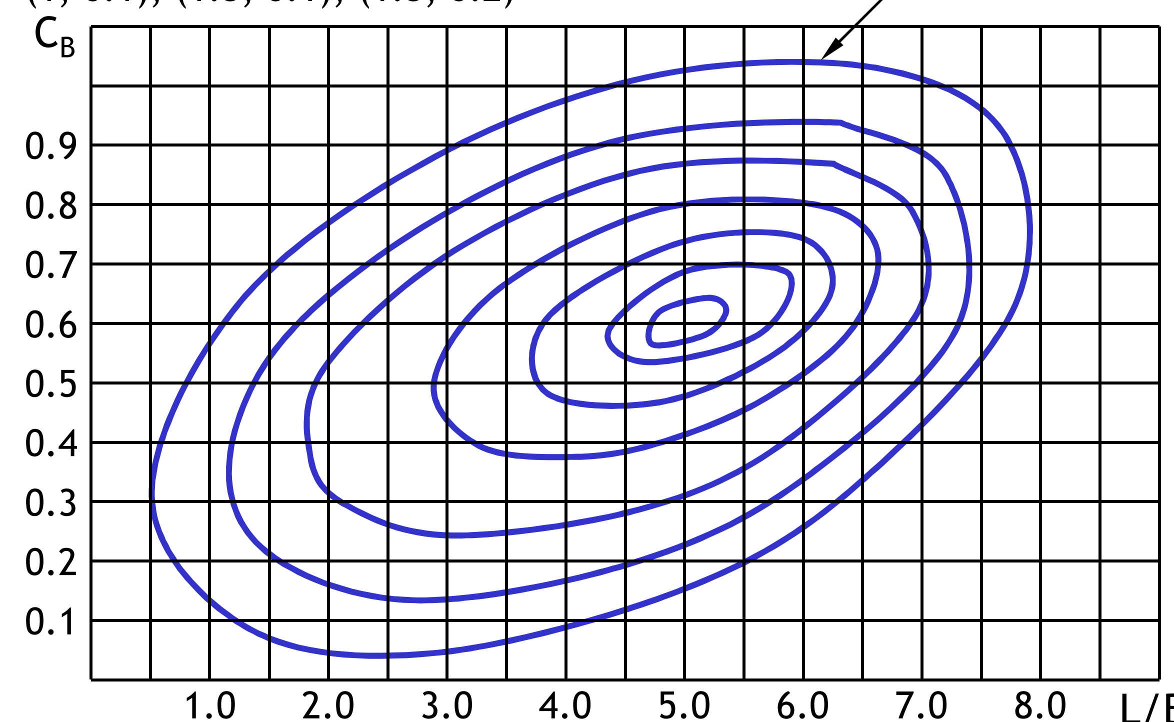
1. 만일 Local Pattern Search를 수행한 후에도 $b^{k+1} = b^k$ 이 되어 어떠한 개선이 이루어지지 않는다면 모든 δ_i를 $\delta_i/2$로 바꾸어 위 과정을 반복한다.
2. 만일 모든 δ_i 가 ε_i보다 작으면 탐사 과정을 마친다.

비제약 최적화 문제

- 어느 선박의 상대적 건조비를 L/B와 C_B의 함수로 다음 그림과 같이 표시할 수 있다고 하면 건조비가 최소가 되는 L/B와 C_B의 값을 Hooke & Jeeves의 탐사법과 Nelder & Mead의 Simplex 방법을 이용하여 최적점을 구하고 그 과정을 그림에 각각 표시하시오.
  - Hooke & Jeeves의 탐사법
    - 출발점: L/B = 1, C_B = 0.1
    - 출발점의 step size: Δ(L/B) = 0.5, Δ(C_B) = 0.1
  - Nelder & Mead의 Simplex 방법
    - 출발 모서리점: (L/B, CB) = (1, 0.1), (1.5, 0.1), (1.5, 0.2)
    - 중지 기준: 0.01

목적 함수의 contour line(f = const.)

미지수 2개인 최적화 문제 ←



비제약 최적화 문제
- Hooke & Jeeves의 직접 탐사법을 이용한 해법(1)

$x_1 = L/B,\ x_2 = C_B$ **라고 놓고 각 탐사 단계를 설명하면 다음과 같다.**

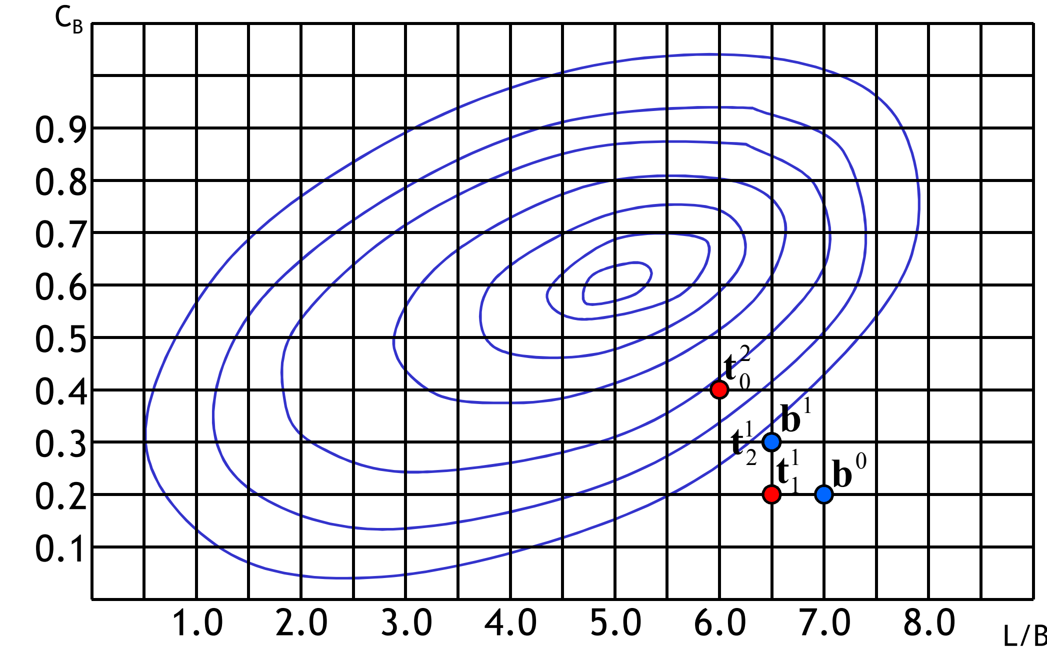
- 단계 1 : 국부 탐사 1

$\mathbf{b}^0 = (7,\ 0.2),\ \Delta x_1 = 0.5,\ \Delta x_2 = 0.1,$

$\mathbf{t}_0^1 = \mathbf{b}^0$

$\mathbf{t}_0^1$에서 $-x_1$방향 $\rightarrow \mathbf{t}_1^1 = (6.5,\ 0.2)$

$\mathbf{t}_1^1$에서 $+x_2$방향 $\rightarrow \mathbf{t}_2^1 = (6.5,\ 0.3)$



국부 탐사가 성공(함수값이 감소) 하였으므로 t_{12}를 새로운 기준점(Base Point)으로 선정한다.

$$\mathbf{b}^1 = \mathbf{t}_2^1$$

- 단계 2 : 전체 탐사 1

$\mathbf{b}^0$와 $\mathbf{b}^1$에서 $\mathbf{t}_0^2 = (6,\ 0.4)$

임시점 $\mathbf{t}_0^2$에서의 함수값이 개선되었으므로 이 점에서부터 다시 국부탐사를 한다.

비제약 최적화 문제
- Hooke & Jeeves의 직접 탐사법을 이용한 해법(2)

- 단계 3 : 국부 탐사 2

$\mathbf{t}_0^2$에서 $-x_1$방향 $\rightarrow \mathbf{t}_1^2 = (5.5,\ 0.4)$

$\mathbf{t}_1^2$에서 $+x_2$방향 $\rightarrow \mathbf{t}_2^2 = (5.5,\ 0.5)$

$$\mathbf{b}^2 = \mathbf{t}_2^2$$

- 단계 4 : 전체 탐사 2

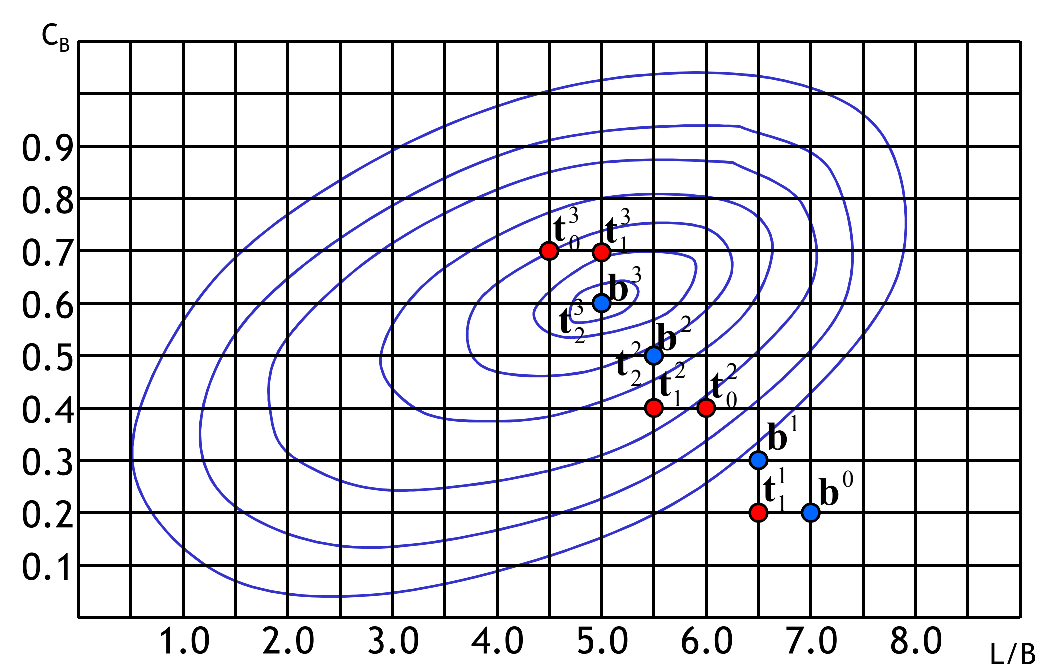
$\mathbf{b}^1$와 $\mathbf{b}^2$에서 $\mathbf{t}_0^3 = (4.5,\ 0.7)$

- 단계 5 : 국부 탐사 3

$\mathbf{t}_0^3$에서 x_1방향 $\rightarrow \mathbf{t}_1^0 = (5,\ 0.7)$

$\mathbf{t}_1^3$에서 $-x_2$방향 $\rightarrow \mathbf{t}_2^3 = (5,\ 0.6)$

$$\mathbf{b}^3 = \mathbf{t}_2^3$$



비제약 최적화 문제
- Hooke & Jeeves의 직접 탐사법을 이용한 해법(3)

● 단계 6 : 전체 탐사 3

$\mathbf{b}^2$와 $\mathbf{b}^3$에서 $\mathbf{t}_0^4 = (4.5,\ 0.7)$에서는 목적 함수값의 감소가 없으므로 $\mathbf{t}_0^4 = \mathbf{b}^3$

● 단계 7 : 국부 탐사 4

$\mathbf{t}_0^4$에서는 x_1방향으로도, x_2방향으로도 목적함수의 감소가 없으므로 $\mathbf{t}_2^4 = \mathbf{t}_1^4 = \mathbf{t}_0^4$

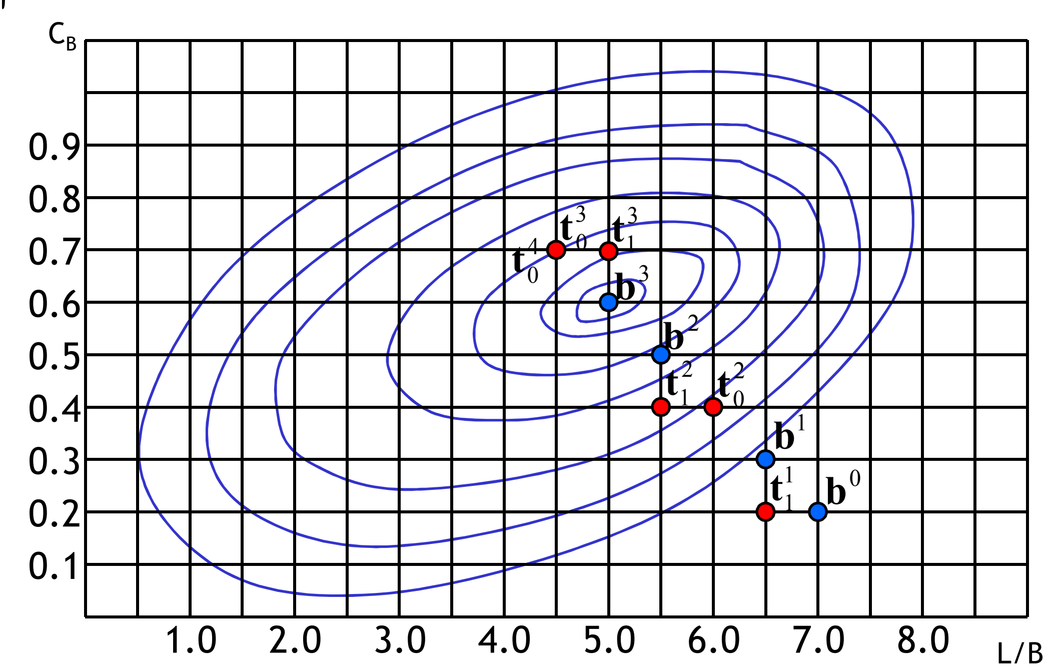
● 단계 8 : 전체 탐사 4

$\mathbf{b}^4 = \mathbf{b}^3 \rightarrow \Delta x_1 = 0.25,\ \Delta x_2 = 0.05,$

$\mathbf{t}_0^5 = \mathbf{b}^4$

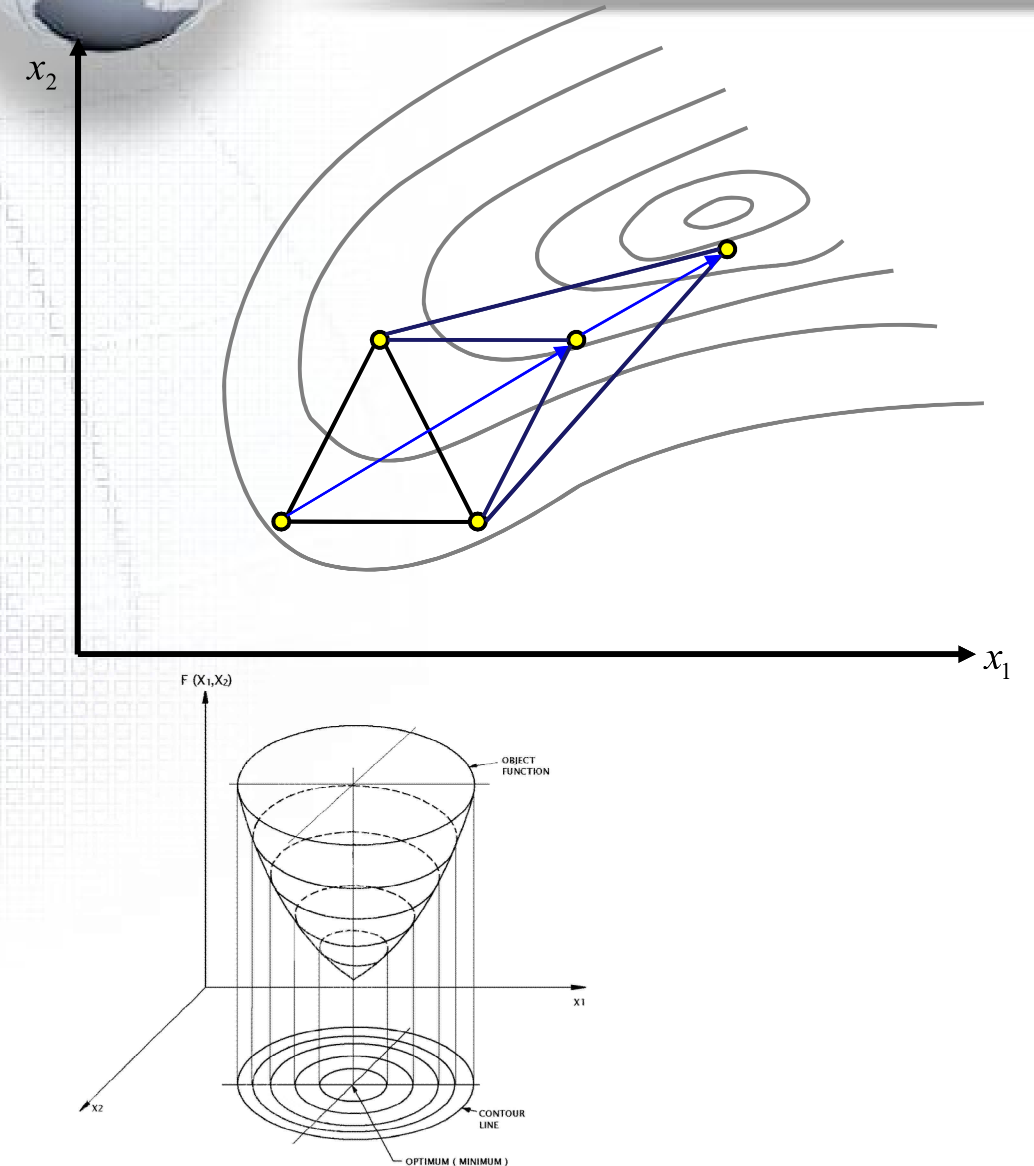
● 단계 9 : 탐사종료

$(x_1, x_2) = (L/B, C_B) = (5.0, 0.6)$까지의 탐사 이후로는 전체탐사와 국부탐사 모두 목적 함수의 감소가 없으므로 탐사가 종료



상대적 건조비가 최소가 되는 최적점은 $L/B = 5.0,\ C_B = 0.6$

2.3 직접 탐사법(Direct Search Method)
- Nelder & Mead 의 Simplex 방법 알고리즘 개요



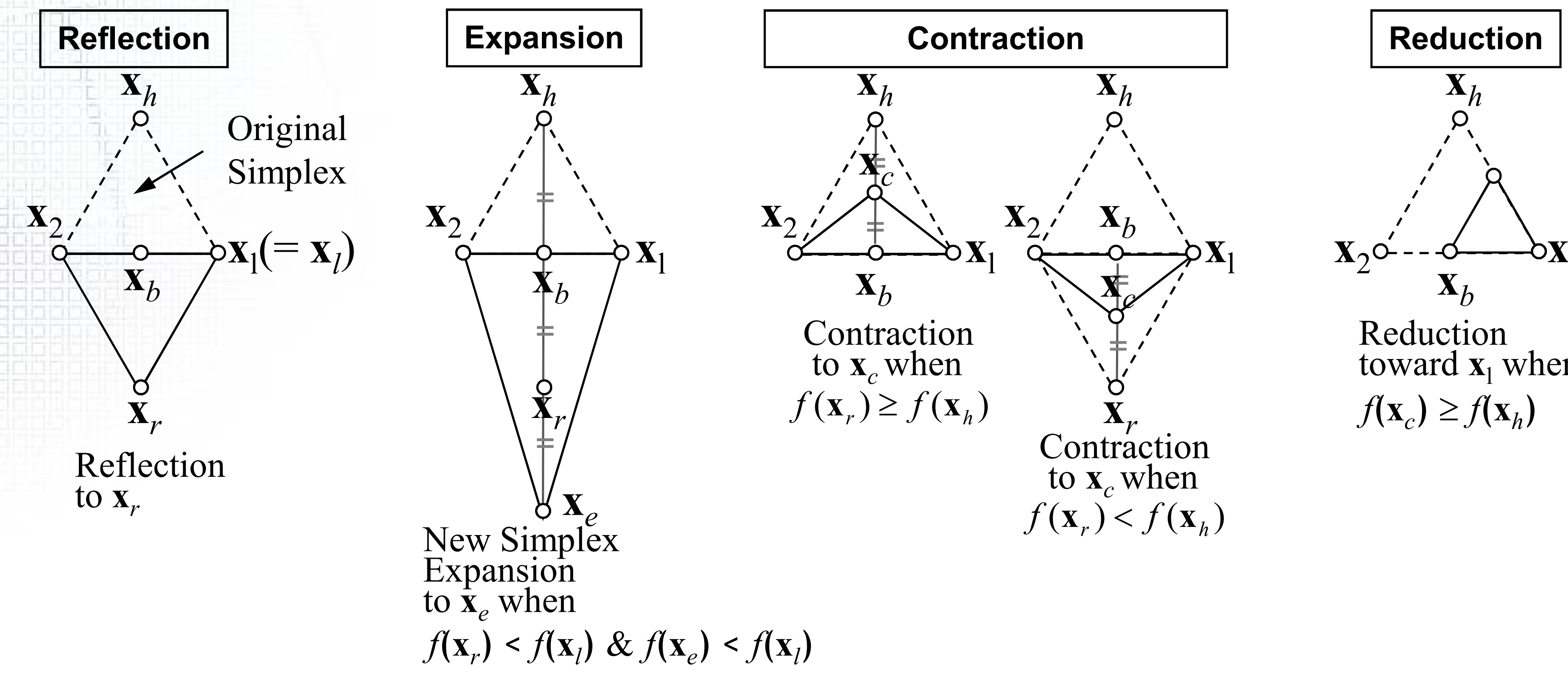
1. 설계 변수의 개수보다 1개가 많은 점을 선택 (설계 변수가 2개이므로 3개의 점 선택)

2. 목적 함수가 개선되는 방향으로 삼각형을 대칭 시켜 목적 함수값을 개선

3. 목적 함수가 개선될 경우 삼각형을 늘이고, 개선이 되지 않으면 줄이면서 최적 설계점을 찾아나감

2.3 직접 탐사법
- Nelder & Mead's Simplex 방법

- n개의 설계 변수를 가진 n차원 문제에서 $(n+1)$개의 모서리를 가진 기하학적 형상(Simplex)의 모양과 위치를 반사(Reflection), 확장(Expansion), 수축(Contraction) 및 감소(Reduction)의 4가지 형태로 계속 변화시키면서 최적점을 찾는 방법



$\mathbf{x}_h$: Simplex point having the largest objective function value

$\mathbf{x}_b$: Center point between $\mathbf{x}_1$ and $\mathbf{x}_2$

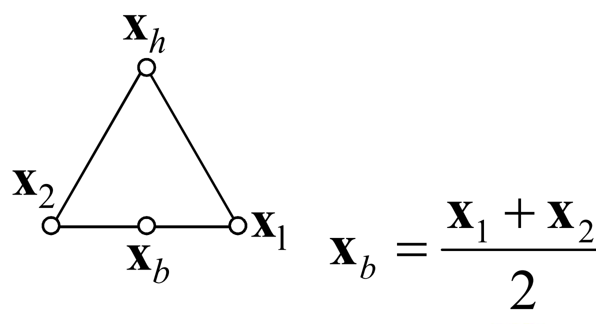
2.3 직접 탐사법
- Nelder & Mead의 Simplex 방법의 알고리즘

- **단계 1 : Simplex 생성 및 목적 함수 값 계산**
  - Simplex의 (n+1)개의 점에서의 목적 함수 f 의 값을 계산한다.

- **단계 2 : 최대 및 최소값 설정**
  - 현재의 Simplex에서 $f(\mathbf{x})$의 값을 최대 및 최소로 하는 점을 각각 $\mathbf{x}_h$, $\mathbf{x}_l$이라 둔다.

- **단계 3 : 중심 계산**
  - $\mathbf{x}_h$를 제외한 모든 $\mathbf{x}_i$에 대한 중심($\mathbf{x}_b$)을 다음과 같이 구하고 이 점에서의 함수 값을 계산한다.

$$\mathbf{x}_b = \frac{1}{n}\sum_{i=1}^{n+1}\mathbf{x}_i \text{(단, } \mathbf{x}_h\text{는 제외)}$$

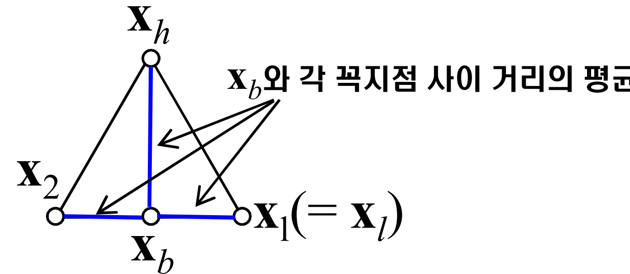


$$\mathbf{x}_b = \frac{\mathbf{x}_1 + \mathbf{x}_2}{2}$$

2.3 직접 탐사법(Direct Search Method) - Nelder & Mead의 Simplex 방법의 알고리즘의 요약(1)

- 단계 4 : 중지 조건

$$\left\{\frac{1}{n+1}\sum_{i=1}^{n+1}[f(\mathbf{x}_i)-f(\mathbf{x}_b)]^2\right\}^{1/2} \le \varepsilon$$

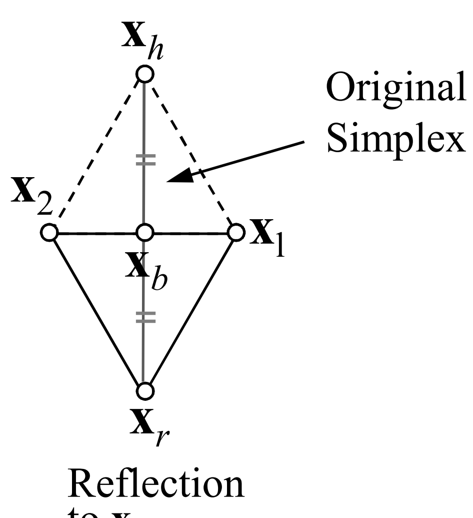


  - 만일 중지 조건을 만족하면 탐사 과정을 중지하고 $\mathbf{x}_l$을 최적 값으로 선택한다. 그렇지 않으면 다음 과정으로 간다.

- 단계 5 : 반사(Reflection)
  - $\mathbf{x}_h$를 $\mathbf{x}_b$에 대해 반사한 점 $\mathbf{x}_r$을 다음과 같이 구한다.

$$\mathbf{x}_r = 2\mathbf{x}_b - \mathbf{x}_h$$

이 점에서의 함수 값 $f(\mathbf{x}_r)$을 계산하고,
다음 조건에 따라 Simplex를 변경시킨다.



- **단계 6 : 확장(Expansion)**
  - 단계 6-1 : $f(\mathbf{x}_r) < f(\mathbf{x}_l)$일 때

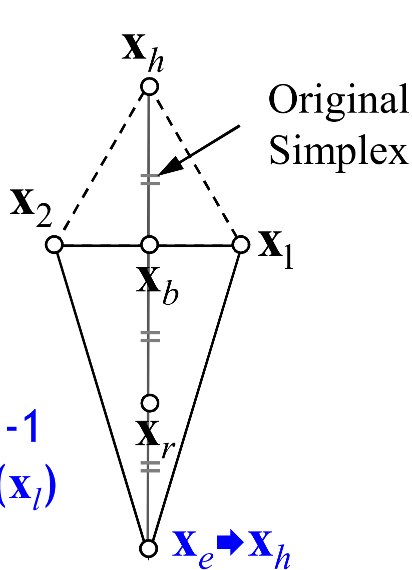
    $\mathbf{x}_b$를 $\mathbf{x}_r$에 대해 반사시켜 확장한 점 $\mathbf{x}_e$를 구한다.

    $$\mathbf{x}_e = 2\mathbf{x}_r - \mathbf{x}_b$$

    이 점에서 함수 값 $f(\mathbf{x}_e)$를 계산하고 이를 $f(\mathbf{x}_l)$과 비교한다.

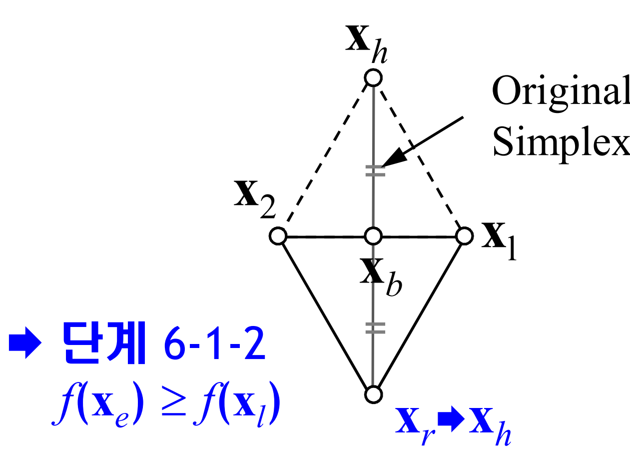
    - 단계 6-1-1 : $f(\mathbf{x}_e) < f(\mathbf{x}_l)$일 때

      $\mathbf{x}_h$를 $\mathbf{x}_e$(확장점)로 대체하고 단계 2로 간다.



    - 단계 6-1-2 : $f(\mathbf{x}_e) \ge f(\mathbf{x}_l)$일 때

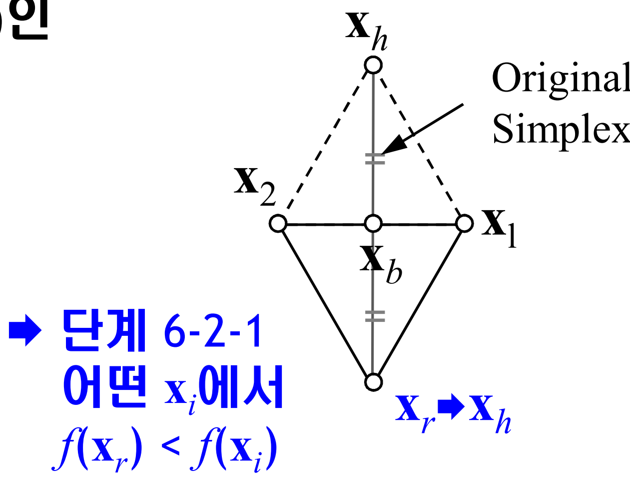
      $\mathbf{x}_h$를 $\mathbf{x}_r$(반사점)로 대체하고 단계 2로 간다.



- Nelder & Mead의 Simplex 방법의 알고리즘의 요약(3)

- 단계 6 : 확장(Expansion)
  - 단계 6-2 : $f(\mathbf{x}_r) \geq f(\mathbf{x}_l)$일 때
    - 단계 6-2-1 : $\mathbf{x}_h$를 제외한 $\mathbf{x}_i$중에서 $f(\mathbf{x}_r) < f(\mathbf{x}_i)$인 경우가 존재할 때

      $\mathbf{x}_h$를 $\mathbf{x}_r$(반사점)로 대체하고 단계 2로 간다.



    - 단계 6-2-2 : $\mathbf{x}_h$를 제외한 $\mathbf{x}_i$중에서 $f(\mathbf{x}_r) < f(\mathbf{x}_i)$인 경우가 존재하지 않을 때

      다음 단계로 간다.

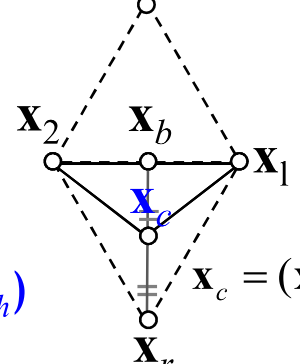
2.3 직접 탐사법
- Nelder & Mead의 Simplex 방법의 알고리즘의 요약(4)

- **단계 7 : 수축**(Contraction)
  - **단계 7-1 :** $f(\mathbf{x}_r) < f(\mathbf{x}_h)$**일 때**

    수축점을 다음과 같이 구하고 이 점에서의 함수 값을 계산한다.

    $$\mathbf{x}_c = (\mathbf{x}_r + \mathbf{x}_b)/2$$

➡ **단계 7-1**
$f(\mathbf{x}_r) < f(\mathbf{x}_h)$



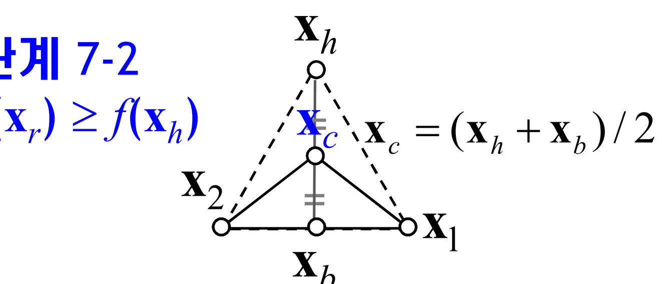
$\mathbf{x}_c = (\mathbf{x}_r + \mathbf{x}_b)/2$

  - **단계 7-2 :** $f(\mathbf{x}_r) \geq f(\mathbf{x}_h)$**일 때**

    수축점을 다음과 같이 구하고 이 점에서의 함수 값을 계산한다.

    $$\mathbf{x}_c = (\mathbf{x}_h + \mathbf{x}_b)/2$$

➡ **단계 7-2**
$f(\mathbf{x}_r) \geq f(\mathbf{x}_h)$



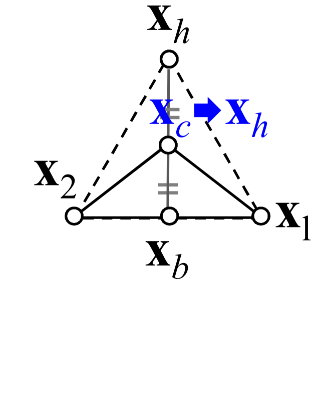
- 단계 8 : 감소(Reduction)
  - 단계 8-1 : $f(\mathbf{x}_c) < f(\mathbf{x}_h)$일 때

    $\mathbf{x}_h$를 $\mathbf{x}_c$(수축점)로 대체하고 단계 2로 간다.

➡ 단계 8-1

$f(\mathbf{x}_c) < f(\mathbf{x}_h)$

또는



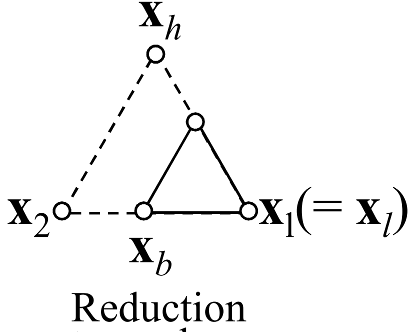
  - 단계 8-2 : $f(\mathbf{x}_c) \geq f(\mathbf{x}_h)$일 때

    Simplex의 점들을 $\mathbf{x}_l$로 향하여 감소시킨 후 단계 2로 간다. 이 때 Simplex의 감소된 점들은 다음과 같이 구한다.

$$\mathbf{x}_i = (\mathbf{x}_i + \mathbf{x}_l)/2$$

➡ 단계 8-2

$f(\mathbf{x}_c) \geq f(\mathbf{x}_h)$



Reduction toward $\mathbf{x}_1$

@SDAL Advanced Ship Design Automation Lab. http://asdal.snu.ac.kr

비제약 최적화 문제
- Nelder & Mead의 Simplex 방법을 이용한 해법(1)

$x_1 = L/B,\ x_2 = C_B$ **라고 놓고 각 탐사 단계를 설명하면 다음과 같다.**

1)삼각형 1 : x_1, x_2, x_3

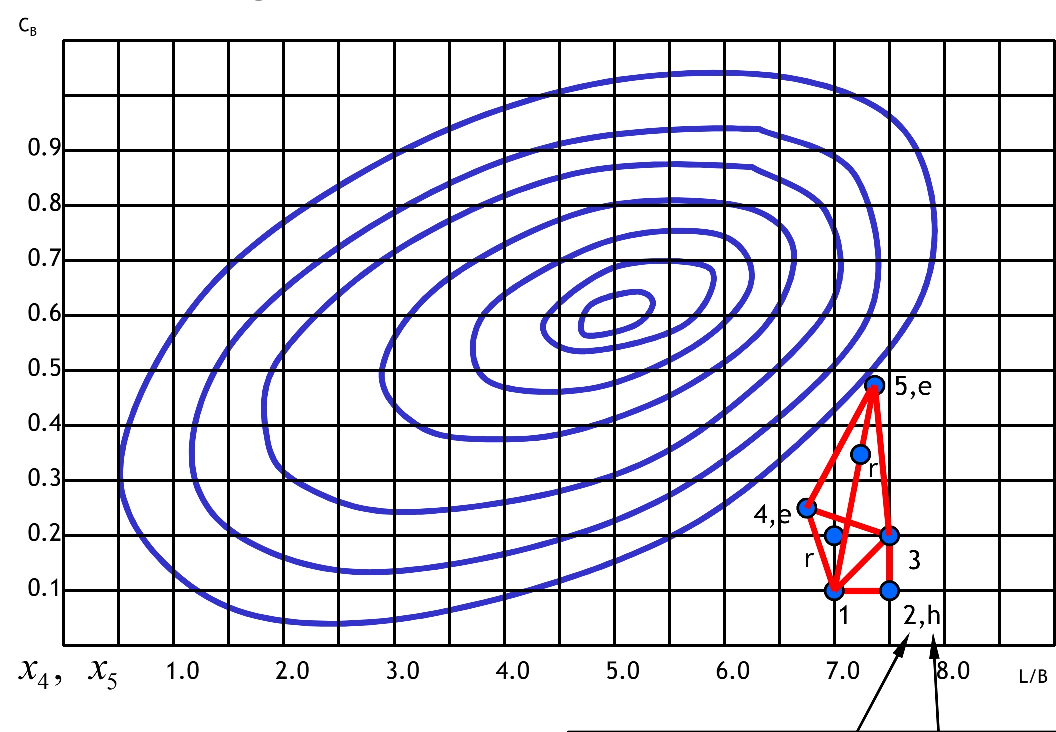
2)x_2가 x_h이므로 x_1, x_3의 중심을 기준으로 대칭 시킨다. $\rightarrow x_r$

x_r이 x_1, x_3보다 모두 작으므로 Expansion을 한다. $\rightarrow x_{4,e}$

$\rightarrow$ 삼각형 2 : x_1, x_3, x_4

3)x_1이 x_h이므로 x_3, x_4의 중심을 기준으로 대칭 시킨다. $\rightarrow x_r$

x_r이 x_3, x_4보다 모두 작으므로 Expansion을 한다. $\rightarrow x_{5,e} \rightarrow$ 삼각형 3 : x_3, x_4, x_5



숫자는 x_i의 인덱스 i를 나타냄

알파벳은 x_i의 종류를 나타냄

h: 삼각형 꼭지점 중 함수 값이 가장 큰 점

r: reflection

e: expansion

c: contraction

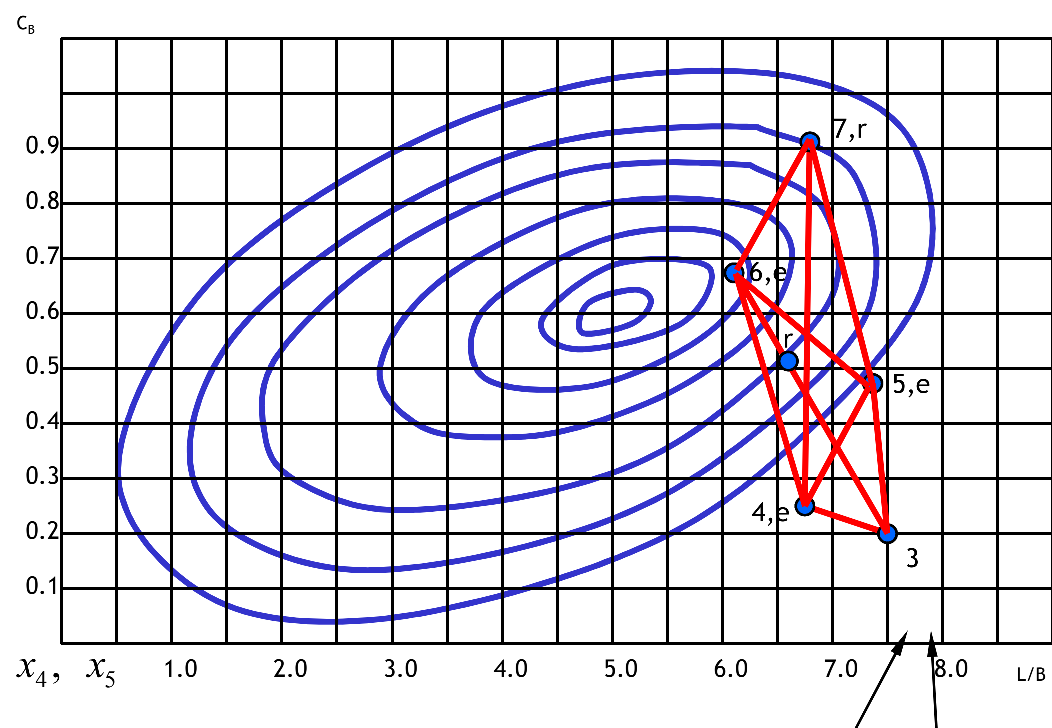
비제약 최적화 문제
- Nelder & Mead의 Simplex 방법을 이용한 해법(1)

$x_1 = L/B,\ x_2 = C_B$ **라고 놓고 각 탐사 단계를 설명하면 다음과 같다.**

1)삼각형 1 : $x_1,\ x_2,\ x_3$

2)x_2가 x_h이므로 x_1, x_3의 중심을 기준으로 대칭 시킨다.$\rightarrow x_r$
x_r이 x_1, x_3보다 모두 작으므로 Expansion을 한다. $\rightarrow x_{4,e}$
$\rightarrow$삼각형 2 : $x_1,\ x_3,\ x_4$

3)x_1이 x_h이므로 x_3, x_4의 중심을 기준으로 대칭 시킨다.$\rightarrow x_r$
x_r이 x_3, x_4보다 모두 작으므로 Expansion을 한다. $\rightarrow x_{5,e} \rightarrow$삼각형 3 : $x_3,\ x_4,\ x_5$



4)x_3이 x_h이므로 x_4, x_5의 중심을 기준으로 대칭 시킨다.$\rightarrow x_r$
x_r이 x_4, x_5보다 모두 작으므로 Expansion을 한다. $\rightarrow x_{5,e} \rightarrow$삼각형 4 : $x_3,\ x_4,\ x_5$

5)x_4가 x_h이므로 x_5, x_6의 중심을 기준으로 대칭 시킨다.$\rightarrow x_{7,r}$
$x_{7,r}$의 값이 x_6보다는 크므로 다음 단계로 간다.$\rightarrow$삼각형 5 : $x_5,\ x_6,\ x_7$

숫자는 x_i의 인덱스 i를 나타냄

알파벳은 x_i의 종류를 나타냄
h: **삼각형 꼭지점 중 함수 값이 가장 큰 점**
r: reflection
e: expansion
c: contraction

비제약 최적화 문제
- Nelder & Mead의 Simplex 방법을 이용한 해법(2)

6) x_5가 x_h이므로 x_6, x_7의 중심을 기준으로 대칭 시킨다. → x_r

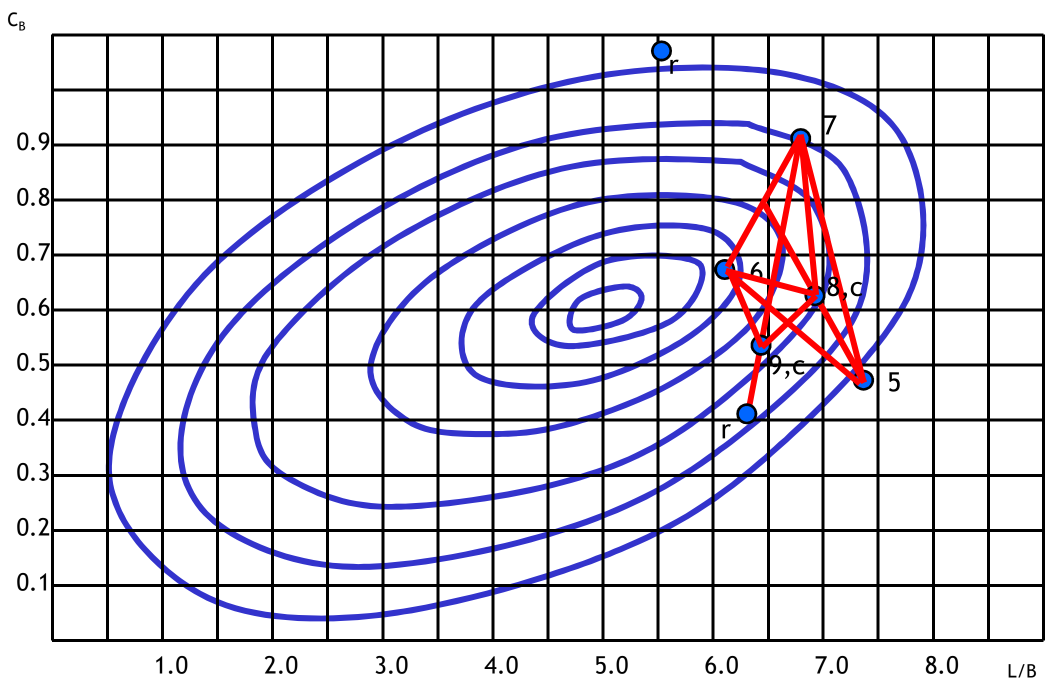
x_r이 x_5, x_6, x_7보다 모두 크므로 x_5방향으로 수축을 한다. → $x_{8,c}$

→ 삼각형 6 : x_6, x_7, x_8

7) x_7이 x_h이므로 x_6, x_8의 중심을 기준으로 대칭 시킨다. → x_r

x_r이 x_6, x_8보다 크고 x_7보다 작으므로 x_r방향으로 수축을 한다. → $x_{9,c}$

→ 삼각형 7 : x_6, x_8, x_9



@SDAL
Advanced Ship Design Automation Lab.
http://asdal.snu.ac.kr

비제약 최적화 문제
- Nelder & Mead의 Simplex 방법을 이용한 해법(2)

6)x_5가 x_h이므로 x_6, x_7의 중심을 기준으로 대칭 시킨다. → x_r

x_r이 x_5, x_6, x_7보다 모두 크므로 x_5방향으로 수축을 한다. → $x_{8,c}$

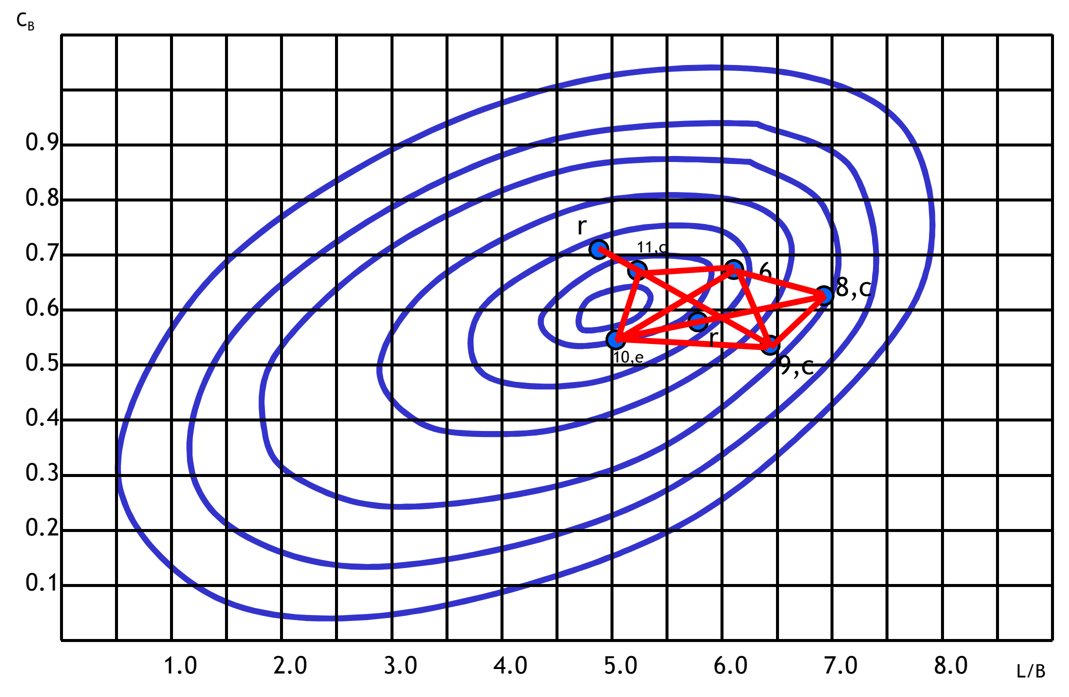
→ 삼각형 6 : x_6, x_7, x_8

7)x_7이 x_h이므로 x_6, x_8의 중심을 기준으로 대칭 시킨다. → x_r

x_r이 x_6, x_8보다 크고 x_7보다 작으므로 x_r방향으로 수축을 한다. → $x_{9,c}$

→ 삼각형 7 : x_6, x_8, x_9

8)x_8이 x_h이므로 x_6, x_9의 중심을 기준으로 대칭 시킨다. → x_r



x_r이 x_6, x_9보다 모두 작으므로 Expansion을 한다. → $x_{10,e}$ → 삼각형 8 : x_6, x_9, x_{10}

9)$x_{9,c}$가 x_h이므로 x_6, x_{10}의 중심을 기준으로 대칭 시킨다. → x_r

x_r이 x_6, x_{10}보다 크고 x_9보다 작으므로 x_r방향으로 수축을 한다. → $x_{11,c}$ → 삼각형 9 : x_6, x_{10}, x_{11}

비제약 최적화 문제
- Nelder & Mead의 Simplex 방법을 이용한 해법(3)

10) x_6이 x_h이므로 x_{10}, x_{11}의 중심을 기준으로 대칭 시킨다. $\rightarrow x_r$

x_r이 x_{10}, x_{11}보다 크고 x_6보다 작으므로 x_r방향으로 수축을 한다. $\rightarrow x_{12,c}$

$\rightarrow$삼각형 10 : x_{10}, x_{11}, x_{12}

$x_1(7, 0.1)$ $x_2(7.5, 0.1)$

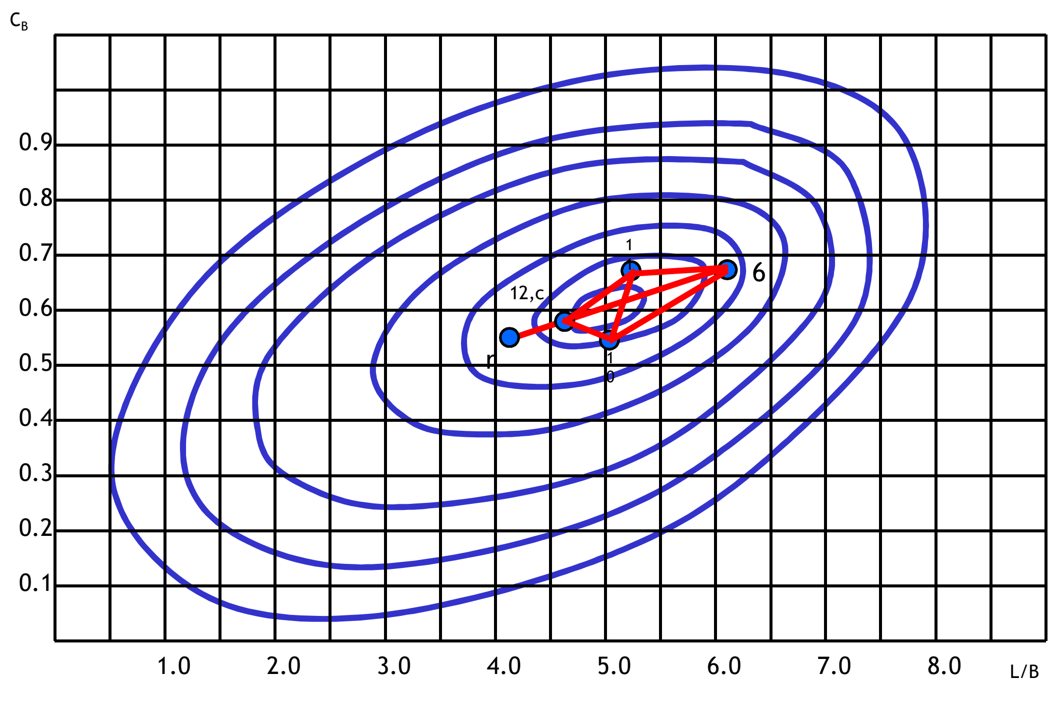
$x_3(7.5, 0.2)$ $x_4(6.75, 0.25)$

$x_5(7.375, 0.475)$ $x_6(6.1875, 0.6875)$

$x_7(6.8125, 0.9125)$ $x_8(6.9375, 0.6375)$

$x_9(6.4375, 0.5375)$ $x_{10}(5.0625, 0.5625)$

$x_{11}(5.21875, 0.66875)$ $x_{12}(4.6171875, 0.5796875)$



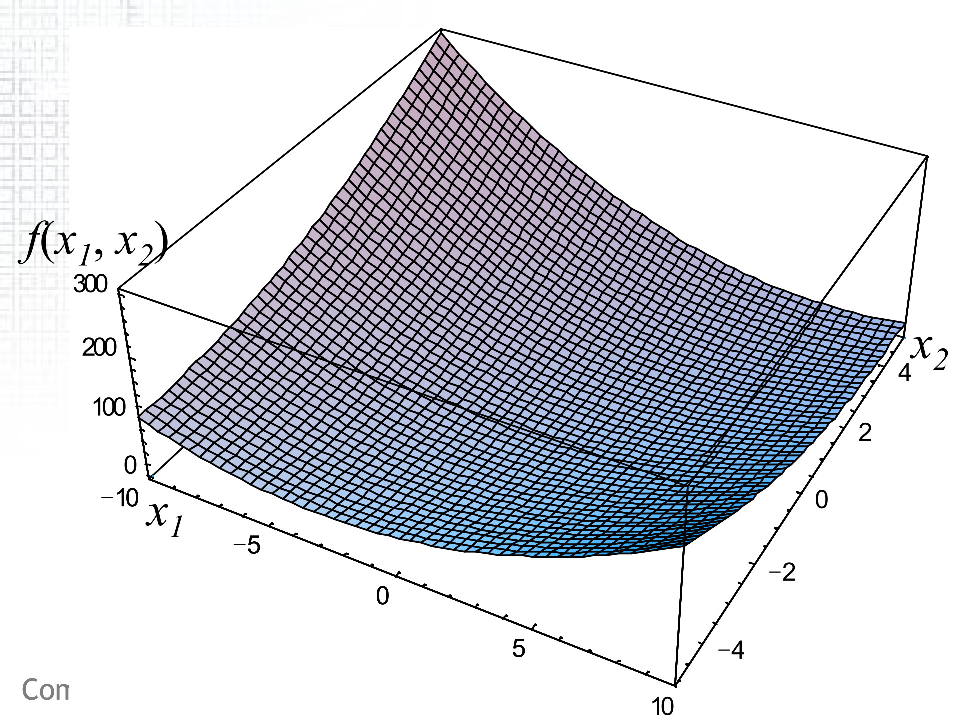
삼각형 10의 완성으로서 탐사를 종료하였으나, 앞에서의 Hooke & Jeeves의 탐사법을 이용한 결과로 얻은 최적 점으로 삼각형이 점차로 접근하게 됨을 확인할 수 있다.

Direct Search Method 프로그램 과제

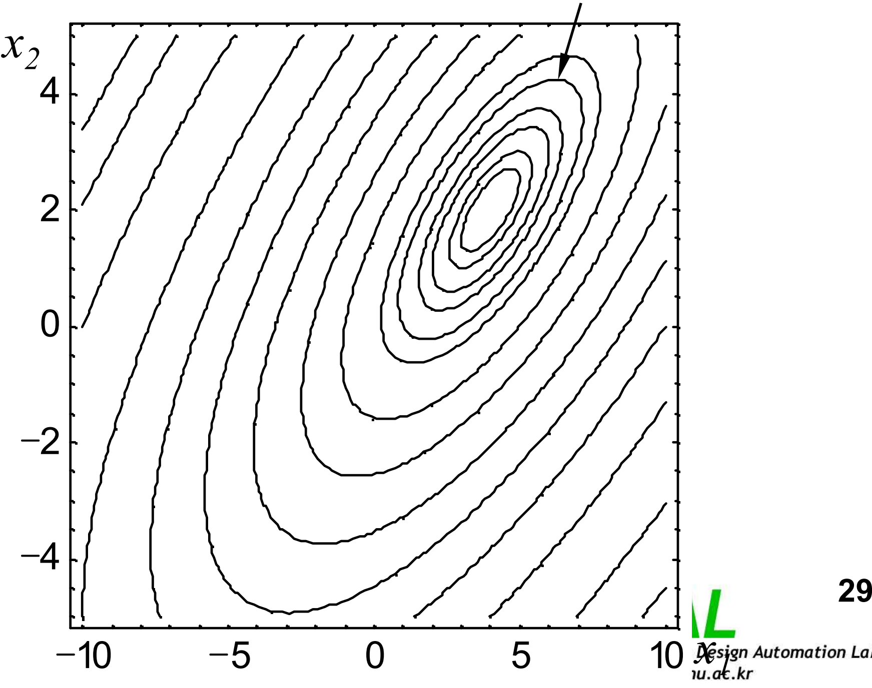
✓ Hooke & Jeeves와 Nelder & Mead 방법을 이용하여
다음 최적화 문제를 풀 수 있는 프로그램을 작성 하시오

Find $x_1(=B/T), x_2(=1/C_B)$

Minimize $f(x_1, x_2) = x_1^2 + 2x_2^2 - 4x_1 - 2x_1x_2 + 10$



목적 함수의 contour line(f = const.)



직접 탐사법 Programming Guide(1)
- Class DirectSearch의 구조 예시

Class DirectSearch

```
class DirectSearch
{
public:
  // constructor
  DirectSearch(void);
  DirectSearch(double* init, double* del, int nNumOfVars, double (*f)(int, double*));
  //초기 설계값, 초기 이동폭, 변수 개수, 함수 입력

  // destructor
  ~DirectSearch(void);

  // member variables
  int m_nNumOfVars;                     // 설계 변수 개수
  double* m_pVarInit;                   // 최초 설계 변수
  double* m_pVarOptimum;                // 결과 설계 변수
  double (*m_fpObjFunc)(int, double*);  // 목적 함수 (함수 포인터)

  // Hooke and Jeeves
  double* m_pDelta;                     // 설계점에 대한 이동 폭
  double* m_pVarPrev;                   // Local Pattern Search 이전 설계 변수
  double* m_pVarNext;                   // Local Pattern Search 이후 설계 변수
  double m_fObjFuncPrev;                // Local Pattern Search 이전 함수 값
  double m_fObjFuncNext;                // Local Pattern Search 이후 함수 값

  // Local Pattern Search를 수행
  // input  : 목적 함수(현재 설계점, 변수의 개수)
  // output : Local Pattern Search 수행 후 설계점에서의 함수 값
  double LocalPatternSearch();

  // Global Pattern Search를 수행
  // input  : 목적 함수(현재 설계점, 변수의 개수)
  // output : Global Pattern Search 수행 후 설계점에서의 함수 값
  double GlobalPatternSearch();

  double HNJ_Optimization();

  // Nelder and Mead
  int m_nVarHigh;                       // 설계 변수 x(high)
  int m_nVarLow;                        // 설계 변수 x(low)
  int m_nSimplexDimension;              // 차수
  double** m_ppVarNnM;
  double* m_pObjFunc;                   // 각 simplex 점에서의 함수 값

  void CalcSimplex();
  double NNM_Optimization();
};
```

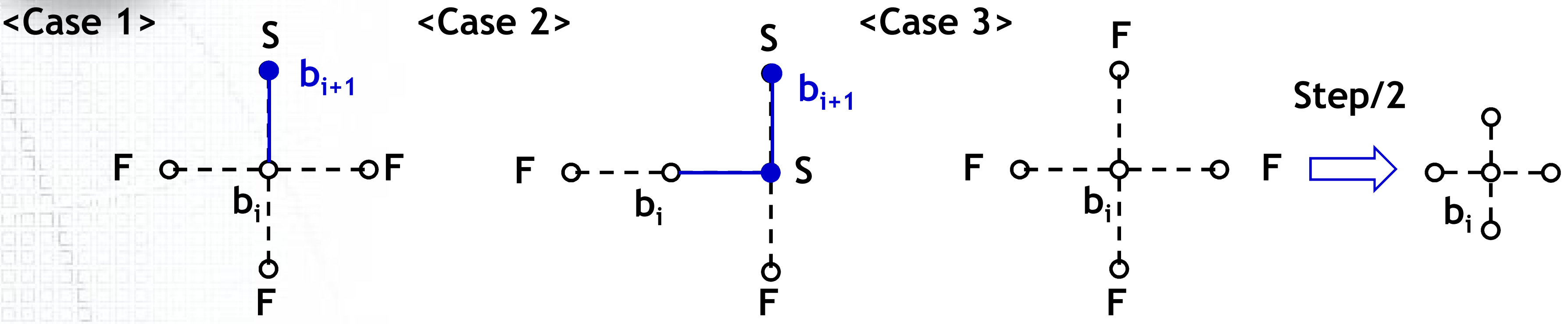
Main

```
int _tmain(int argc, _TCHAR* argv[])
{
  int numOfVariable = 2;
  double optimum;
  double init[2];
  double del[2];

  //시작점 설정
  init[0] = -1;
  init[1] = -2;
  //Search Step 결정
  del[0] = 0.1;
  del[1] = 0.1;

  //test1이라는 객체 선언
  DirectSearch test1(init, del, numOfVariable, f1);
  //Hooke & Jeeves 방법으로 최적화 문제를 풂
  optimum = test1.HNJ_Optimization();

  //결과 출력
  printf("optimum value: %f \n", optimum);
  printf("optimum point: ");
  for(int i = 0; i < test1.m_numOfVariable-1; i++)
  {
    printf("%f, ", test1.m_Variable_optimum[i]);
  }
  printf("%f \n",
test1.m_Variable_optimum[test1.m_numOfVariable-1]);

  return 0;
}
```

직접 탐사법 Programming Guide (2)

- 구현 예: Hook & Jeeves 방법 중 Local Pattern Search

- Local Pattern Search (F: Fail, S: Success)

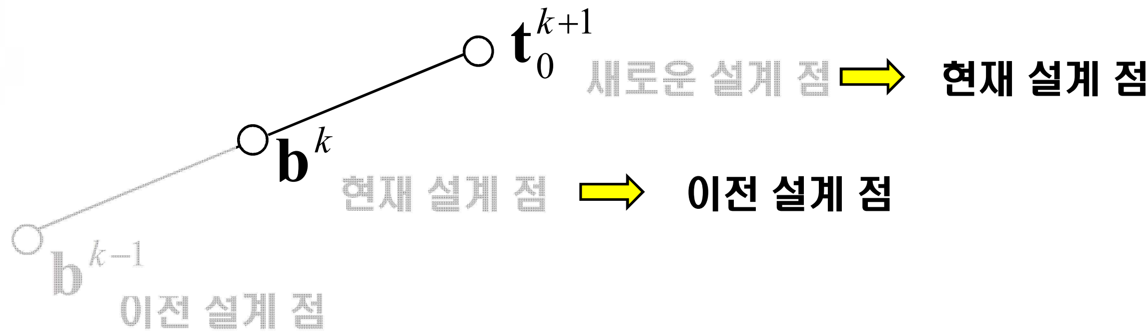



```
// [input]    del: 설계점에 대한 이동폭, x: 현재의 설계점,
//        fmin_pre: 이전 설계점에서의 함수값, variables_no: 설계 변수의 수
// [output] 개선된 설계점과 이 점에서의 목적 함수값
double LocalPatternSearch(double* del, double* x, double fmin_pre, int variables_no)
{
    for(int i = 0; i< m_numOfVariable; i++)
    {
        // 설계점을 이동폭 만큼 증가시켜 함수값을 구한다.
        // 기존의 함수값과 비교하여 기존 함수값보다 작으면 기존 함수값을 교체
        // 기존의 함수값보다 크면 설계점을 이동폭만큼 감소시켜 함수값을 구한다.
        // 기존의 함수값과 비교하여 기존 함수값보다 작으면 기존함수값을 교체
        // 기존 함수값이 작으면 임시설계변수를 기존의 설계변수로 다시되돌림
    }
    return fmin_pre;
}
```

직접 탐사법 Programming Guide (3)

- 구현 예: Hook & Jeeves 방법 중 Global Pattern Move

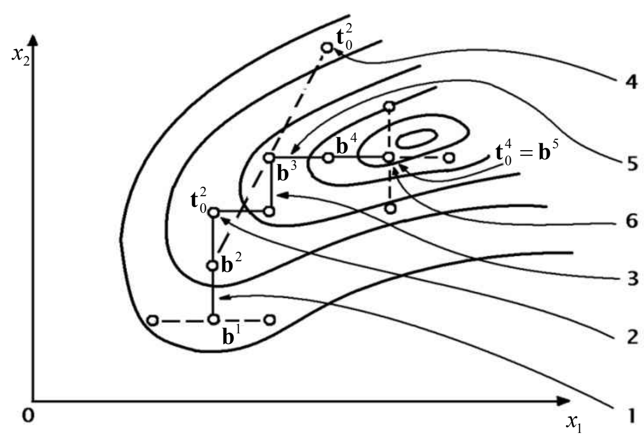
- Global Pattern Move




```
//[input]    b_before: 이전의 설계점,                b_now        : 현재의 설계점,
//           fmin_pre: 이전 설계점에서의 함수값, variables_no : 설계 변수의 수
//[output]    Global Pattern Search 수행 후 설계점, 새로운 설계점 에서의 함수값,
double GlobalPatternMove(double* b_before, double* b_now, double fmin_pre, int variables_no
                         double* x)
{
    // 이전의 설계점을 현재의 설계점에 대칭시켜 새로운 설계점을 구한다.
    // 새로운 설계점의 함수값이 작으면 최소값을 새 설계점의 함수값으로 변경한다.
    // 새로운 설계점의 함수값이 크면 새로운 설계점을 현재의 설계점으로 되돌린다.

    // global move 가 끝나고 현재 설계점을 이전 설계점으로, 새로운 설계점을 현재 설계점으로 변경한다.

    return fmin_pre;
}
```

직접 탐사법 Programming Guide(4)

- 구현 예: Local Pattern Search, Global Pattern Move를 이용한 Hooke & Jeeves 방법



```
double DirectSearch::HNJ_Optimization()
{
    do {
        do {
            // Local Pattern Search를 수행한다.
            m_minimum_after = LocalPatternSearch();

            // 함수값 개선이 이루어지지 않으면 이동폭을 1/2로 줄인다.
            // HNJ가 완전히 끝나는 조건을 체크(모든 step size가 epsilon 이하)
        } while();  // HNJ가 끝나거나 Local serch가 끝나는 조건을 체크
        m_minimum_before = m_minimum_after = GlobalPatternMove();
    } while(); // 모든 step size가 epsilon 이하인 조건을 체크
    return m_minimum_after;
}
```


@SDAL Advanced Ship Design Automation Lab. http://asdal.snu.ac.kr

Term Project #6

Golden section search method, Direct search method

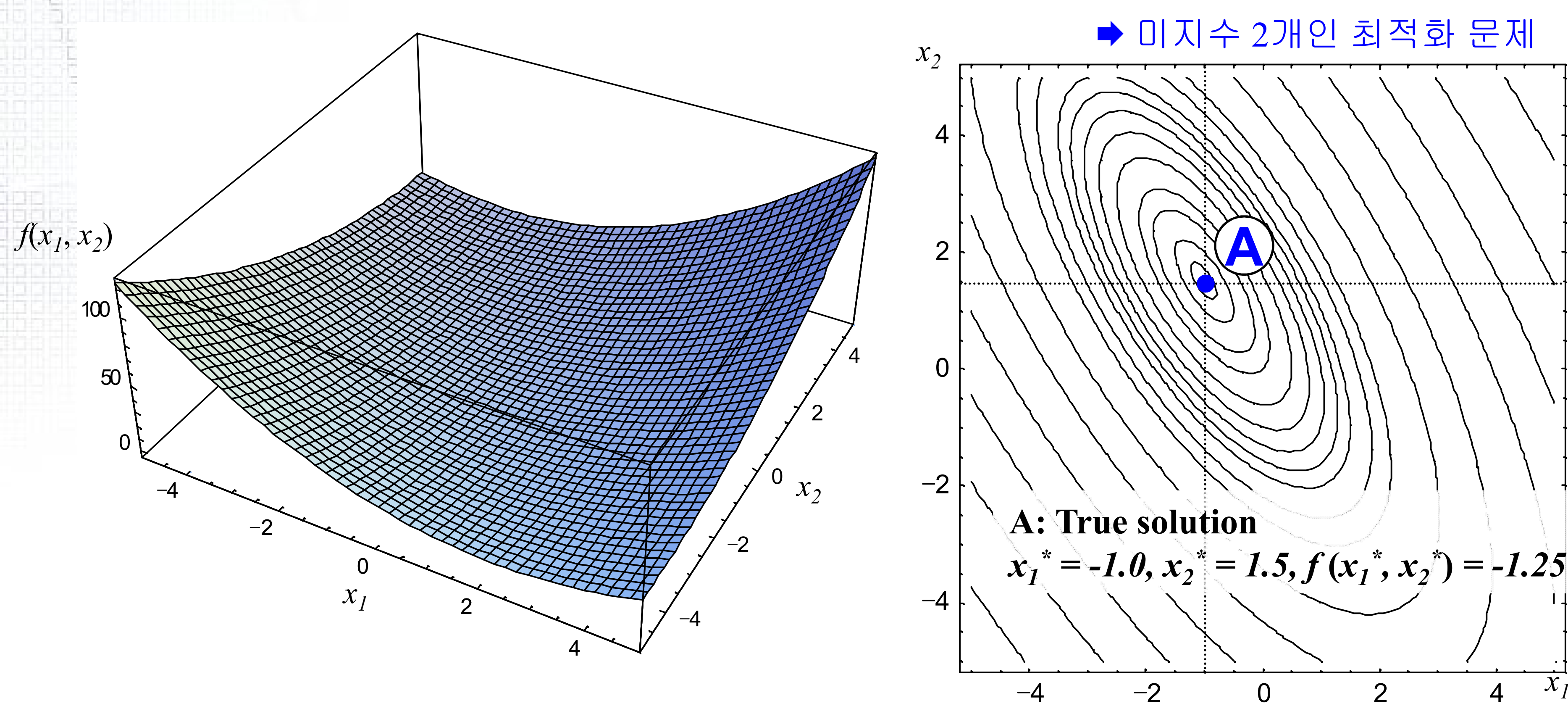
Advanced
Ship
Design
Automation
Laboratory

비제약 최적화 문제 #1

- **황금분할법, 직접탐사법을 이용하여** 2**변수 함수의 최소점을 구하시오. 단, 시작점 x**$^{(0)}$ = (0, 0), convergence tolerance ε = 0.001**이며, x**$^{(3)}$**까지 구하시오.** (**황금분할법 사용 시 탐색 방향은** (-1.0, 1.5))

$$\text{Minimize} \quad f(x_1, x_2) = x_1 - x_2 + 2x_1^2 + 2x_1 x_2 + x_2^2$$

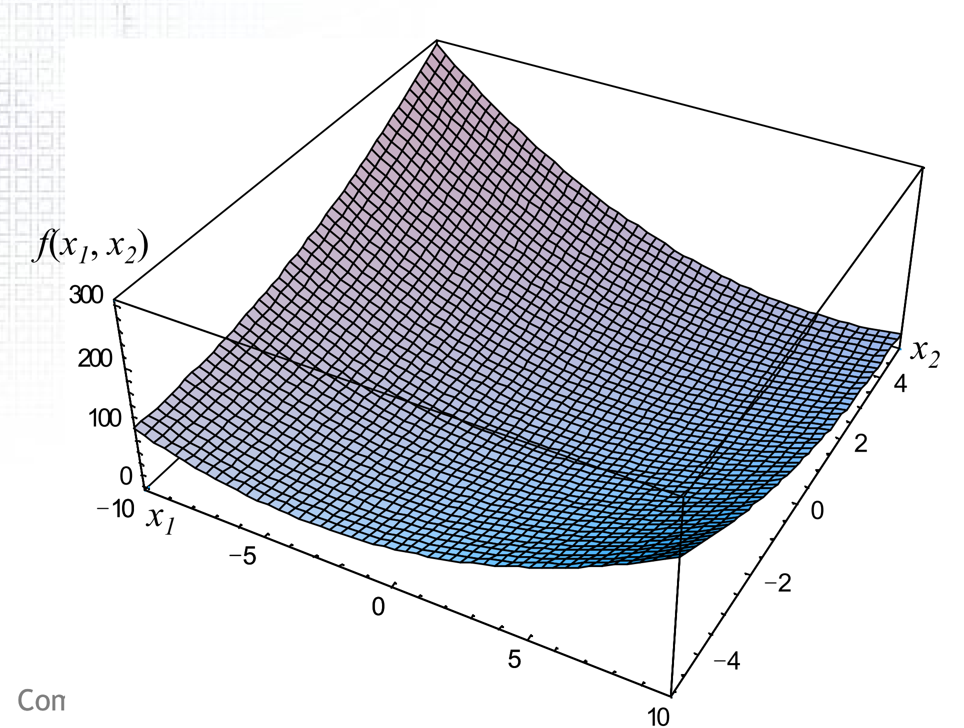
➡ 미지수 2개인 최적화 문제

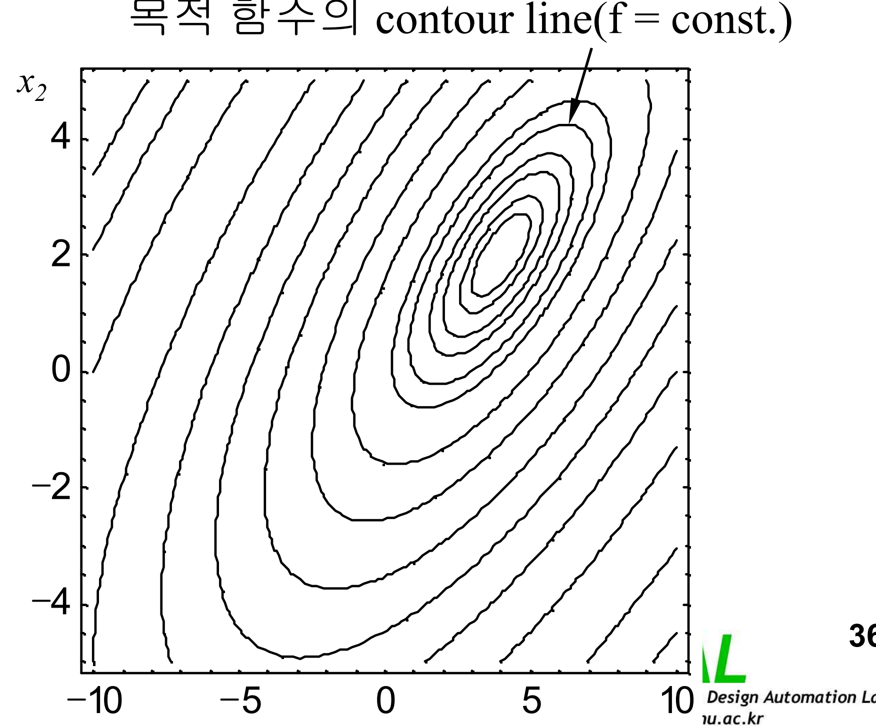


비제약 최적화 문제 #2

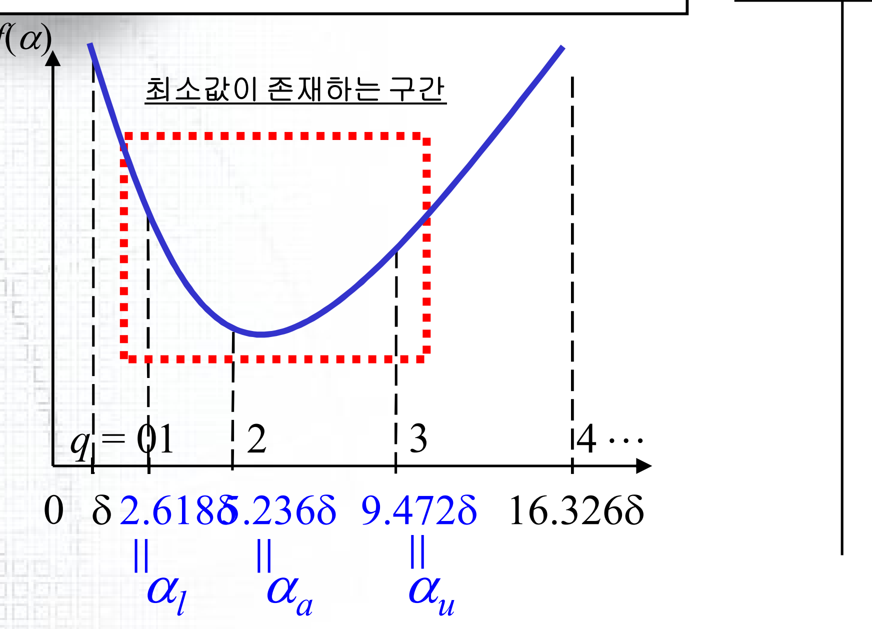
- **황금분할법, 직접탐사법을 이용하여 2변수 함수의 최소점을 구하시오. 단, 시작점 $\mathbf{x}^{(0)}$ = (0, 0), convergence tolerance ε = 0.001이며, $\mathbf{x}^{(3)}$까지 구하시오. (황금분할법 사용 시 탐색 방향은 (1.0, 1.0))**

$$\textit{Minimize} \quad f(x_1, x_2) = x_1^2 + 2x_2^2 - 4x_1 - 2x_1x_2 + 10$$

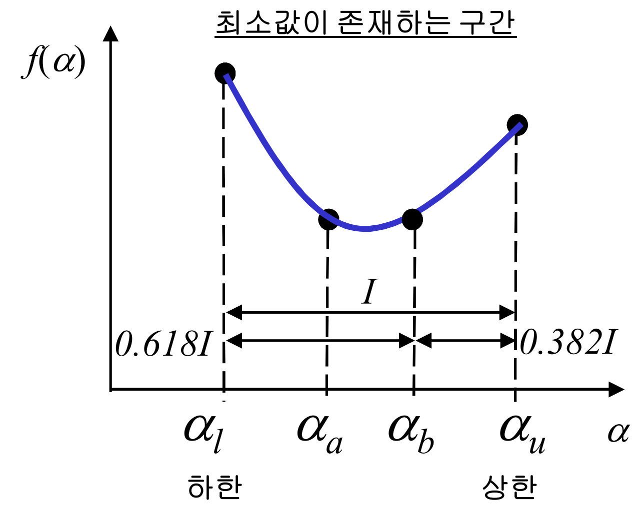




황금 분할법 Programming Guide
- 1변수 함수

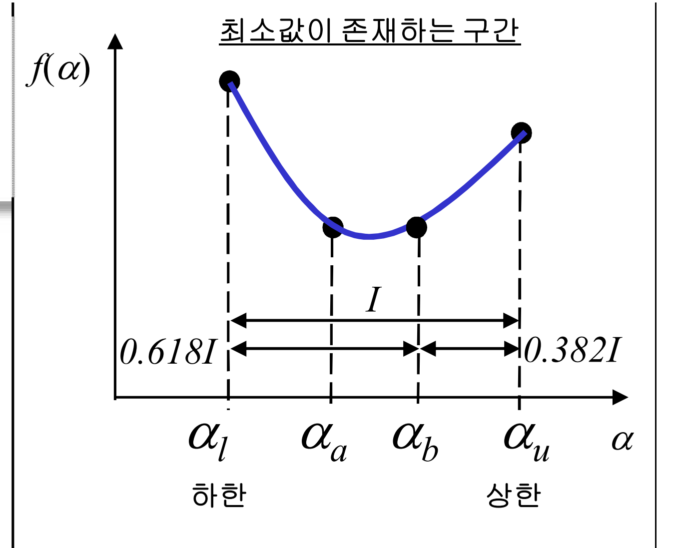


2단계 : $f(\alpha_a)$ 과 $f(\alpha_b)$ 를 계산
(단, $\alpha_b = \alpha_l + 0.618I, \alpha_a = \alpha_l + 0.382I$)




```
//Input : 초기 위치(initial_x), 증분값 (delta)
//output : 최소값이 존재하는 구간의 x좌표 (x[0],x[1],x[2]
void findMinValueExistSection(double initial_x, double delta, double *x)
{
    //초기값에서 1.618δ배씩 증가시키면서 최소 구간을 탐색함
    //x[1]보다 x[2]가 클 때 종료 시킴
}
```

황금 분할법 Programming Guide
- 1변수 함수



3단계 : $f(\alpha_a)$ 과 $f(\alpha_b)$ 를 비교

① $f(\alpha_a) < f(\alpha_b)$

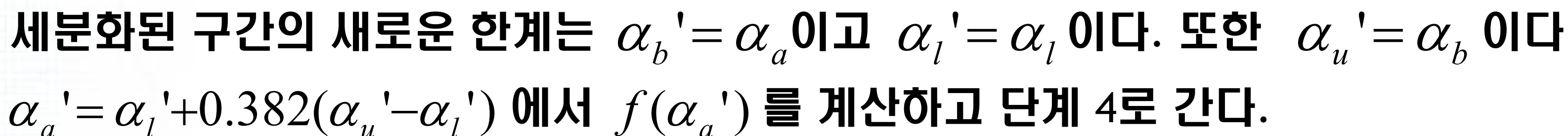
최적점 α^*는 α_l과 α_b 사이에 있다.

세분화된 구간의 새로운 한계는 $\alpha_b' = \alpha_a$이고 $\alpha_l' = \alpha_l$ 이다. 또한 $\alpha_u' = \alpha_b$ 이다.

$\alpha_a' = \alpha_l' + 0.382(\alpha_u' - \alpha_l')$ 에서 $f(\alpha_a')$ 를 계산하고 단계 4로 간다.

② $f(\alpha_a) > f(\alpha_b)$

최적점 α^*는 α_a과 α_u 사이에 있다.



세분화된 구간의 새로운 한계는 $\alpha_l' = \alpha_a$이고 $\alpha_a' = \alpha_b$이다. 또한 $\alpha_u' = \alpha_u$ 이다.

$\alpha_b' = \alpha_l' + 0.618(\alpha_u' - \alpha_l')$ 에서 $f(\alpha_b')$ 를 계산하고 단계 4로 간다.

③ $f(\alpha_a) = f(\alpha_b)$

$\alpha_l = \alpha_a, \alpha_u = \alpha_b$ 라 두고 단계 4로 간다.

4단계 : 구간 $(I' = \alpha_u' - \alpha_l')$ 의 길이가 Tolerance보다 작으면 종료한다. 아닐 경우 2단계로 돌아간다.

황금 분할법 Programming Guide

- 1차원 함수에 대한 황금분할법 함수 예시

```
// [Input]
//    x[0]: 최소값이 존재하는 영역의 하한
//    x[2]: 최소값이 존재하는 영역의 상한
//    x[1]: x[0] < x[1] < x[2]인 동시에 f(x[0]) > f(x[1]) and f(x[2]) > f(x[1])인 점
// [Output]
//    xmin: f를 최소로 하는 점과 이 점에서의 목적 함수값

double GoldenSectionSearch(double *x, double (*f)(double), double *xmin)
{
     double TOLERANCE = 1.0e-6;
     double f1, f2, a0, a1, a2, a3;
     a0 = x[0];                        a3 = x[2];

     if (fabs(x[2] – x[1]) > fabs(x[1] - x[0])) {
            a1 = x[1]; a2 = x[1] + (1.0 - 0.618) * (x[2] - x[1]);  }
     else { a2 = x[1]; a1 = x[1] - (1.0 - 0.618) * (x[1] - x[0]);  }

     f1 = (*f)(a1);    f2 = (*f)(a2);

     while (fabs(a3 - a0) > TOLERANCE ) {
            if (f2 < f1) {       a0 = a1;    a1 = a2;    a2 = 0.618 * a1 + (1.0 - 0.618) * a3;
                                 f1 = f2;    f2 = (*f)(a2);                                 }
            else {      a3 = a2;    a2 = a1;    a1 = 0.618 * a2 + (1.0 - 0.618) * a0;
                        f2 = f1;    f1 = (*f)(a1);                                          }
     }

     if (f1 < f2) {    *xmin = a1;                      return f1; }
     else {            *xmin = a2;                      return f2; }
}
```

황금 분할법 Programming Guide

- 황금분할법 Program 예시

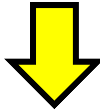
```
#include <stdio.h>
#include <math.h>

double f1(double x);
double f2(double x);
double f3(double x);
void FindSection(double x_start, double x_delta, double (*ObjFunc)(double),
double *x);
double GoldenSectionSearch(double *x, double (*f)(double), double *xmin);

int main()
{
  double init_x = 0;
  double delta = 1;
  double *x = new double [3];
  double *xmin = 0;
  double f_min;

  //f1
  FindSection(init_x,delta,f1,x);
  f_min = GoldenSectionSearch(x,f1,xmin);

  //f2
  FindSection(init_x,delta,f2,x);
  f_min = GoldenSectionSearch(x,f2,xmin);

  //f3
  FindSection(init_x,delta,f3,x);
  f_min = GoldenSectionSearch(x,f3,xmin);

  return 0;
}
```

```
//f(x)=x^2
double f1(double x)
{
      return x*x;
}

//f(x)=sin x
double f2(double x)
{
     return sin(x);
}

//f(x)=x^3-x^2+x-1
double f3(double x)
{
       return pow(x,3) - pow(x,2) + x - 1;
}
```

황금 분할법의 확장
- 목적 함수가 다변수 함수일 때 탐색 방향이 주어질 경우

	목적 함수 식	탐색 방향	시작 위치
1변수 함수	double (*f)(double x) 하나의 변수만 입력	1변수 함수이므로 탐색 방향은 이미 주어져 있음	double x_start
다변수 함수	double (*f)(double *x, int n) 여러 개의 변수를 입력 받고, 변수의 개수도 입력	탐색 방향을 Vector 형태로 주어야 함	double *x_start

```
double f(double x);
void FindSection(double x_start, double x_delta, double (*f)(double), double *x);
double GoldenSectionSearch(double *section, double (*f)(double), double *xmin);
```



```
double f(double *x, int n);
void FindSection(double *x_start, double *x_delta, double (*f)(double*, int), double **x);
double GoldenSectionSearch(double **section, double (*f)(double*, int), double *xmin);
```

Direct Search Method 프로그램 Guide(1)

- Class DirectSearch의 구조 예시

Class DirectSearch

```
class DirectSearch
{
public:
  // constructor
  DirectSearch(void);
  DirectSearch(double* init, double* del, int nNumOfVars, double (*f)(int, double*));
  //초기 설계값, 초기 이동폭, 변수 개수, 함수 입력

  // destructor
  ~DirectSearch(void);

  // member variables
  int m_nNumOfVars;                      // 설계 변수 개수
  double* m_pVarInit;                    // 최초 설계 변수
  double* m_pVarOptimum;                 // 결과 설계 변수
  double (*m_fpObjFunc)(int, double*);   // 목적 함수 (함수 포인터)

  // Hooke and Jeeves
  double* m_pDelta;                      // 설계점에 대한 이동 폭
  double* m_pVarPrev;                    // Local Pattern Search 이전 설계 변수
  double* m_pVarNext;                    // Local Pattern Search 이후 설계 변수
  double m_fObjFuncPrev;                 // Local Pattern Search 이전 함수 값
  double m_fObjFuncNext;                 // Local Pattern Search 이후 함수 값

  // Local Pattern Search를 수행
  // input  : 목적 함수(현재 설계점, 변수의 개수)
  // output : Local Pattern Search 수행 후 설계점에서의 함수 값
  double LocalPatternSearch();

  // Global Pattern Search를 수행
  // input  : 목적 함수(현재 설계점, 변수의 개수)
  // output : Global Pattern Search 수행 후 설계점에서의 함수 값
  double GlobalPatternSearch();

  double HNJ_Optimization();

  // Nelder and Mead
  int m_nVarHigh;                        // 설계 변수 x(high)
  int m_nVarLow;                         // 설계 변수 x(low)
  int m_nSimplexDimension;               // 차수
  double** m_ppVarNnM;
  double* m_pObjFunc;                    // 각 simplex 점에서의 함수 값

  void CalcSimplex();
  double NNM_Optimization();
};
```

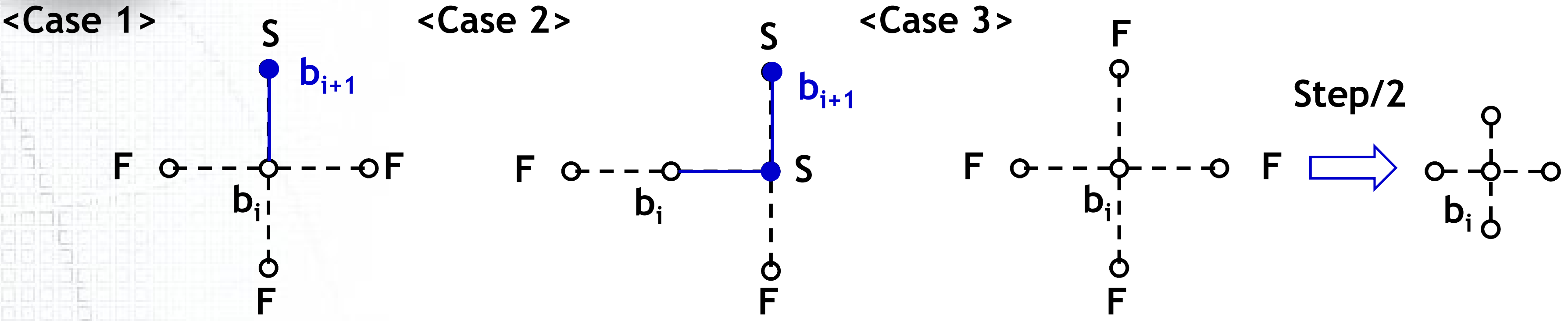
Main

```
int _tmain(int argc, _TCHAR* argv[])
{
  int numOfVariable = 2;
  double optimum;
  double init[2];
  double del[2];

  //시작점 설정
  init[0] = -1;
  init[1] = -2;
  //Search Step 결정
  del[0] = 0.1;
  del[1] = 0.1;

  //test1이라는 객체 선언
  DirectSearch test1(init, del, numOfVariable, f1);
  //Hooke & Jeeves 방법으로 최적화 문제를 품
  optimum = test1.HNJ_Optimization();

  //결과 출력
  printf("optimum value: %f \n", optimum);
  printf("optimum point: ");
  for(int i = 0; i < test1.m_numOfVariable-1; i++)
  {
    printf("%f, ", test1.m_Variable_optimum[i]);
  }
  printf("%f \n",
test1.m_Variable_optimum[test1.m_numOfVariable-1]);

  return 0;
}
```

Direct Search Method 프로그램 Guide(3)

- 구현 예: Hook & Jeeves 방법 중 Local Pattern Search

- Local Pattern Search (F: Fail, S: Success)

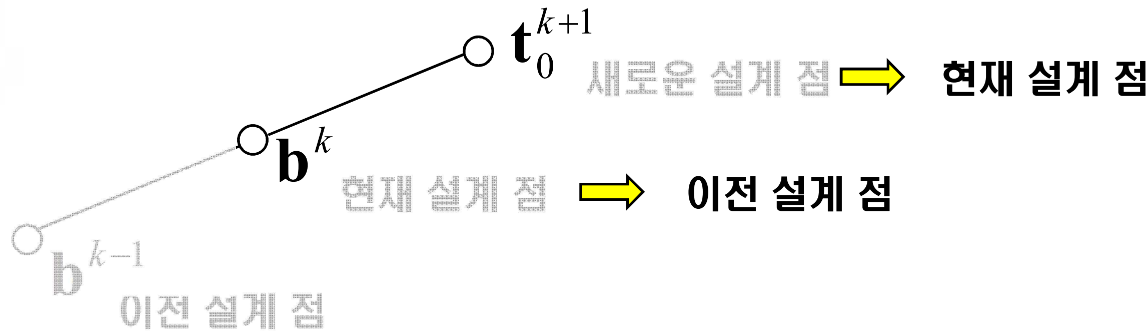



```
// [input]    del: 설계점에 대한 이동폭, x: 현재의 설계점,
//         fmin_pre: 이전 설계점에서의 함수값, variables_no: 설계 변수의 수
// [output] 개선된 설계점과 이 점에서의 목적 함수값
double LocalPatternSearch(double* del, double* x, double fmin_pre, int variables_no)
{
    for(int i = 0; i< m_numOfVariable; i++)
    {
        // 설계점을 이동폭 만큼 증가시켜 함수값을 구한다.
        // 기존의 함수값과 비교하여 기존 함수값보다 작으면 기존 함수값을 교체
        // 기존의 함수값보다 크면 설계점을 이동폭만큼 감소시켜 함수값을 구한다.
        // 기존의 함수값과 비교하여 기존 함수값보다 작으면 기존함수값을 교체
        // 기존 함수값이 작으면 임시설계변수를 기존의 설계변수로 다시되돌림
    }
    return fmin_pre;
}
```

Direct Search Method 프로그램 Guide(4)

- 구현 예: Hook & Jeeves 방법 중 Local Pattern Search

- **Global Pattern Move**




```
//[input]   b_before: 이전의 설계점, b_now: 현재의 설계점,
//          fmin_pre: 이전 설계점에서의 함수값, variables_no: 설계 변수의 수
//[output] Global Pattern Search 수행 후 설계점, 새로운 설계점 에서의 함수값,
double GlobalPatternMove(double* b_before, double* b_now, double fmin_pre, int variables_no
                         double* x)
{
        //이전의 설계점을 현재의 설계점에 대칭시켜 새로운 설계점을 구한다.
        //새로운 설계점의 함수값이 작으면 최소값을 새 설계점의 함수값으로 변경한다.
        //새로운 설계점의 함수값이 크면 새로운 설계점을 현재의 설계점으로 되돌린다.

        //global move 가끝나고 현재 설계점을 이전설계점으로,
        //새로운 설계점을 현재설계점으로 변경한다.

        return fmin_pre;
}
```

Direct Search Method 프로그램 Guide(5)

- 구현 예: Local Pattern Search와 Global Pattern Move를 이용한 Hook & Jeeves 방법

```
double DirectSearch::HNJ_Optimization()
{
        do{

                do{
                        //Local Pattern Search를수행한다.
                        m_minimum_after = LocalPatternSearch();
                        //함수값개선이이루어지지않으면이동폭을1/2로줄인다.
                        //HNJ가 완전히 끝나는 조건을 체크(모든 step size가 epsilon 이하)
                }while();//HNJ가 끝나거나 Local serch가 끝나는 조건을 체크
                m_minimum_before = m_minimum_after = GlobalPatternMove();
        }while(); //모든 step size가 epsilon 이하인 조건을 체크
        return m_minimum_after;
}
```

직접 탐사법 Programming Guide(1)

- Class DirectSearch의 구조 예시

Class DirectSearch

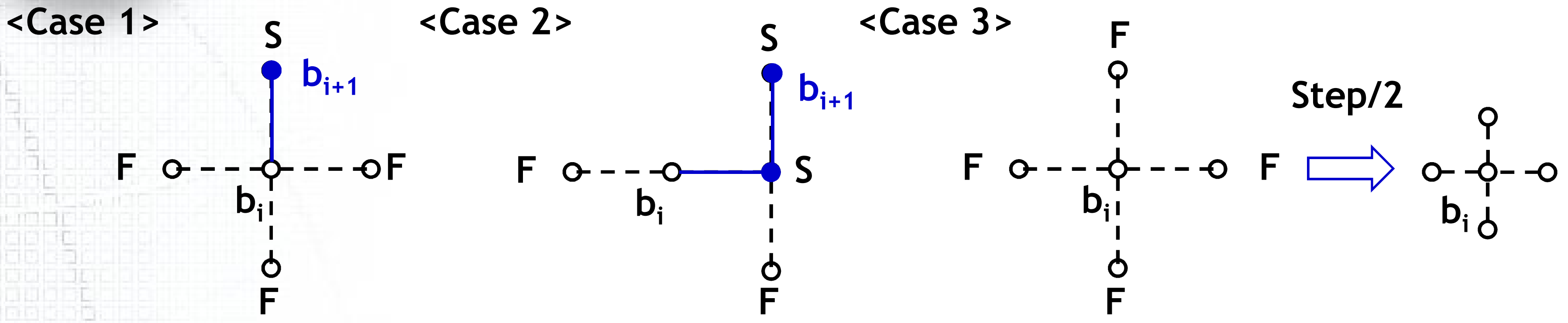
```
class DirectSearch
{
public:
  // constructor
  DirectSearch(void);
  DirectSearch(double* init, double* del, int nNumOfVars, double (*f)(int, double*));
  //초기 설계값, 초기 이동폭, 변수 개수, 함수 입력

  // destructor
  ~DirectSearch(void);

  // member variables
  int m_nNumOfVars;                      // 설계 변수 개수
  double* m_pVarInit;                    // 최초 설계 변수
  double* m_pVarOptimum;                 // 결과 설계 변수
  double (*m_fpObjFunc)(int, double*);  // 목적 함수 (함수 포인터)

  // Hooke and Jeeves
  double* m_pDelta;                      // 설계점에 대한 이동 폭
  double* m_pVarPrev;                    // Local Pattern Search 이전 설계 변수
  double* m_pVarNext;                    // Local Pattern Search 이후 설계 변수
  double m_fObjFuncPrev;                 // Local Pattern Search 이전 함수 값
  double m_fObjFuncNext;                 // Local Pattern Search 이후 함수 값

  // Local Pattern Search를 수행
  // input  : 목적 함수(현재 설계점, 변수의 개수)
  // output : Local Pattern Search 수행 후 설계점에서의 함수 값
  double LocalPatternSearch();

  // Global Pattern Search를 수행
  // input  : 목적 함수(현재 설계점, 변수의 개수)
  // output : Global Pattern Search 수행 후 설계점에서의 함수 값
  double GlobalPatternSearch();

  double HNJ_Optimization();

  // Nelder and Mead
  int m_nVarHigh;                        // 설계 변수 x(high)
  int m_nVarLow;                         // 설계 변수 x(low)
  int m_nSimplexDimension;               // 차수
  double** m_ppVarNnM;
  double* m_pObjFunc;                    // 각 simplex 점에서의 함수 값

  void CalcSimplex();
  double NNM_Optimization();
};
```

Main

```
int _tmain(int argc, _TCHAR* argv[])
{
  int numOfVariable = 2;
  double optimum;
  double init[2];
  double del[2];

  //시작점 설정
  init[0] = -1;
  init[1] = -2;
  //Search Step 결정
  del[0] = 0.1;
  del[1] = 0.1;

  //test1이라는 객체 선언
  DirectSearch test1(init, del, numOfVariable, f1);
  //Hooke & Jeeves 방법으로 최적화 문제를 품
  optimum = test1.HNJ_Optimization();

  //결과 출력
  printf("optimum value: %f \n", optimum);
  printf("optimum point: ");
  for(int i = 0; i < test1.m_numOfVariable-1; i++)
  {
    printf("%f, ", test1.m_Variable_optimum[i]);
  }
  printf("%f \n",
test1.m_Variable_optimum[test1.m_numOfVariable-1]);

  return 0;
}
```

직접 탐사법 Programming Guide (2)
- 구현 예: Hook & Jeeves 방법 중 Local Pattern Search

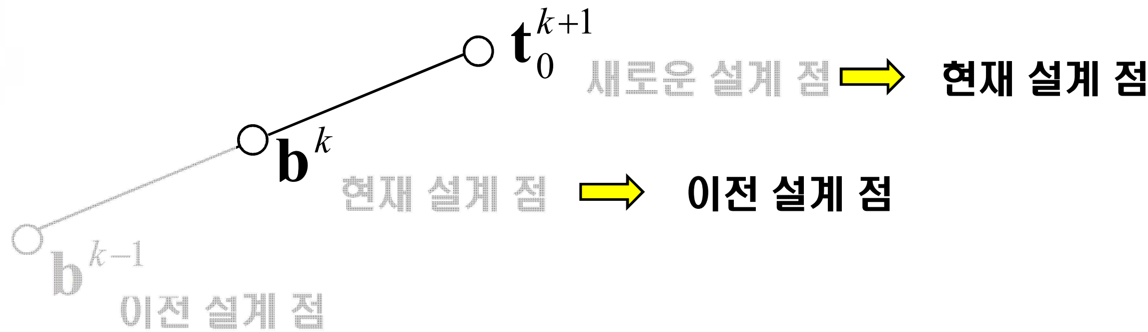
- Local Pattern Search (F: Fail, S: Success)




```
// [input]   del: 설계점에 대한 이동폭, x: 현재의 설계점,
//        fmin_pre: 이전 설계점에서의 함수값, variables_no: 설계 변수의 수
// [output] 개선된 설계점과 이 점에서의 목적 함수값
double LocalPatternSearch(double* del, double* x, double fmin_pre, int variables_no)
{
    for(int i = 0; i< m_numOfVariable; i++)
    {
        // 설계점을 이동폭 만큼 증가시켜 함수값을 구한다.
        // 기존의 함수값과 비교하여 기존 함수값보다 작으면 기존 함수값을 교체
        // 기존의 함수값보다 크면 설계점을 이동폭만큼 감소시켜 함수값을 구한다.
        // 기존의 함수값과 비교하여 기존 함수값보다 작으면 기존함수값을 교체
        // 기존 함수값이 작으면 임시설계변수를 기존의 설계변수로 다시되돌림
    }
    return fmin_pre;
}
```

직접 탐사법 Programming Guide (3)
- 구현 예: Hook & Jeeves 방법 중 Global Pattern Move

- Global Pattern Move

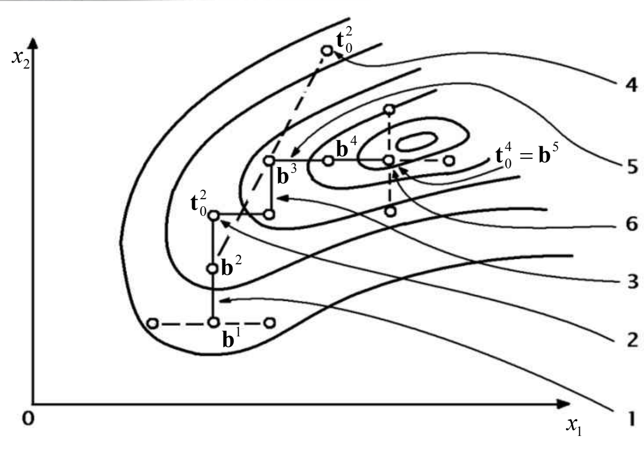



```
//[input]    b_before: 이전의 설계점,                    b_now        : 현재의 설계점,
//           fmin_pre: 이전 설계점에서의 함수값, variables_no : 설계 변수의 수
//[output]    Global Pattern Search 수행 후 설계점, 새로운 설계점 에서의 함수값,
double GlobalPatternMove(double* b_before, double* b_now, double fmin_pre, int variables_no
                                    double* x)
{
    // 이전의 설계점을 현재의 설계점에 대칭시켜 새로운 설계점을 구한다.
    // 새로운 설계점의 함수값이 작으면 최소값을 새 설계점의 함수값으로 변경한다.
    // 새로운 설계점의 함수값이 크면 새로운 설계점을 현재의 설계점으로 되돌린다.

    // global move 가 끝나고 현재 설계점을 이전 설계점으로, 새로운 설계점을 현재 설계점으로 변경한다.

    return fmin_pre;
}
```


@SDAL Advanced Ship Design Automation Lab. http://asdal.snu.ac.kr

직접 탐사법 Programming Guide(4)

- 구현 예: Local Pattern Search, Global Pattern Move를 이용한 Hooke & Jeeves 방법




```
double DirectSearch::HNJ_Optimization()
{
    do {
        do {
            // Local Pattern Search를 수행한다.
            m_minimum_after = LocalPatternSearch();

            // 함수값 개선이 이루어지지 않으면 이동폭을 1/2로 줄인다.
            // HNJ가 완전히 끝나는 조건을 체크(모든 step size가 epsilon 이하)
        } while();  // HNJ가 끝나거나 Local serch가 끝나는 조건을 체크
        m_minimum_before = m_minimum_after = GlobalPatternMove();
    } while(); // 모든 step size가 epsilon 이하인 조건을 체크
    return m_minimum_after;
}
```


@SDAL Advanced Ship Design Automation Lab. http://asdal.snu.ac.kr