Σ

AMPL：数理計画のためのモデリング言語

第6章　複合集合と添字

対・三つ組・組の集合と、集合で添字づけられた集合の集まりで複雑な構造を表現する

AGENDA

本章の内容

6.1 順序対の集合

対象をコンマで区切り括弧で囲んで書く順序対と、その集合

6.2 順序対の部分集合とスライス

within で許される対だけを扱い、スライスで和をとる

6.3 より長い組の集合

三つ組・四つ組など任意長の組へ自然に拡張

6.4 組の集合に対する演算

union・cross・setof など複合集合の演算子

6.5 集合の添字づけられた集まり

他の集合で添字づけられた集合 AREA{p} の考え方

複合集合とは何か

ほとんどの線形計画モデルは、複数の集合の要素の組合せにわたる添字づけを含む。対・三つ組・四つ組など、対象の「組」を要素とする集合を **複合集合（compound sets）** と呼ぶ。順序対の集合の宣言と使用から本章を始める。

- 順序が意味を持つ
- ("STL","PITT") と ("PITT","STL") は別物
- 同じ対象が第1・第2要素の両方に現れてよい

```
// 順序対の書き方
("PITT","STL")
("bands",5)
(3,101)

// 順序対の集合（リテラル）
{("PITT","STL"),("PITT","FRE"),
  ("PITT","DET"),("CLEV","FRE")}

// 数値の対の集合
{(1,1),(1,2),(1,3),(2,1),(2,2),
 (2,3),(3,1),(3,2),(3,3)}
```

二つの集合からできる全対の集合

- 二つの与えられた集合からできるすべての順序対の集合が頻繁に現れる。
- 添字式では {ORIG, DEST} または {i in ORIG, j in DEST} と書く。
- 対の集合を明示的に定義すると、発地・着地の対が都市間の「リンク」であることを強調できる。

- **添字が二つ必要**

revenue[p,t] や Sell[p,t] のように、対の順序どおりに添字を書く。第1添字は PROD、第2添字は 1..T に対応する。

```
// 輸送モデル（図3-1a）
{ORIG, DEST}
{i in ORIG, j in DEST}

// 多期間生産モデル（図4-4）
{PROD, 1..T}
{p in PROD, t in 1..T}
```

```
// リンクとして明示的に定義
set LINKS = {ORIG,DEST};

param cost {LINKS} >= 0;
var Trans {LINKS} >= 0;
```

図6-1　全対を使った輸送モデル

```
set ORIG;   # origins
set DEST;   # destinations

set LINKS = {ORIG,DEST};

param supply {ORIG} >= 0;   # amounts available at origins
param demand {DEST} >= 0;   # amounts required at destinations

   check: sum {i in ORIG} supply[i] = sum {j in DEST} demand[j];

param cost {LINKS} >= 0;    # shipment costs per unit
var Trans {LINKS} >= 0;     # units to be shipped

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];

subject to Supply {i in ORIG}:
   sum {j in DEST} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
   sum {i in ORIG} Trans[i,j] = demand[j];
```

transp2.mod

- LINKS = {ORIG,DEST} は ORIG と DEST の全対。
- sum の添字はダミー添字を対の形 (i,j) で定義しなければならない。
- {k in LINKS} のように単一のダミー添字で走らせるのは誤り。
- 図3-1a と同じ問題を、リンクという概念で書き直したもの。

使える対だけを部分集合として宣言する

- すべての発地からすべての着地へ輸送できるとは限らない。
- 費用と輸送量を、使える発地・着地の対だけで添字づけたい。
- LINKS を全対の部分集合として宣言する。

```
// 最低限（次元だけ指定）
set LINKS dimen 2;
```

```
// より良い：部分集合として宣言
set LINKS within {ORIG,DEST};
```

意図が明確になり、データの誤りも検出できる

cost・Trans・目的関数の宣言はそのまま

```
param cost {LINKS} >= 0;
var Trans {LINKS} >= 0;

sum {(i,j) in LINKS}
   cost[i,j] * Trans[i,j];
```

- cost[i,j]・Trans[i,j] は LINKS の要素としてデータで与えられた対についてのみ定義される。
- 和の式は指定された対だけにわたって取られる。

制約の書き方 —— スライスで和をとる

元の書き方はもう使えない

```
subject to Supply {i in ORIG}:
   sum {j in DEST} Trans[i,j]
      = supply[i];
```

```
error processing constraint Supply['GARY']:
  invalid subscript Trans['GARY','FRA']
```

LINKS にない対まで合計しようとしてしまう

正しい書き方 —— LINKS でスライス

```
subject to Supply {i in ORIG}:
   sum {(i,j) in LINKS} Trans[i,j]
      = supply[i];
```

- i がすでに定義済みの文脈では、{(i,j) in LINKS} は「(i,j) が LINKS に属するすべての j」の集合と解釈される。
- 書ける同義の詳細形：

```
sum {j in DEST: (i,j) in LINKS} Trans[i,j]
```

図6-2a　選ばれた対を使った輸送モデル

```
set ORIG;   # origins
set DEST;   # destinations

set LINKS within {ORIG,DEST};

param supply {ORIG} >= 0;   # amounts available at origins
param demand {DEST} >= 0;   # amounts required at destinations

   check: sum {i in ORIG} supply[i] = sum {j in DEST} demand[j];

param cost {LINKS} >= 0;    # shipment costs per unit
var Trans {LINKS} >= 0;     # units to be shipped

minimize Total_Cost:
   sum {(i,j) in LINKS} cost[i,j] * Trans[i,j];

subject to Supply {i in ORIG}:
   sum {(i,j) in LINKS} Trans[i,j] = supply[i];

subject to Demand {j in DEST}:
   sum {(i,j) in LINKS} Trans[i,j] = demand[j];
```

transp3.mod / .dat

- 図6-1との違いは within の1行と、Supply/Demand の sum だけ。
- データは GARY・CLEV・PITT から一部の着地への疎なリンクのみ。
- 第9章で複合集合のデータ指定方法を詳しく扱う。

表で見るスライス ―― {(i,j) in LINKS}

	FRA	DET	LAN	WIN	STL	FRE	LAF
GARY		×	×		×		×
CLEV	×	×	×	×	×		×
PITT	×			×	×	×	

行＝発地、列＝着地。×は *LINKS に属する対（疎な部分集合）*

i, j とも未定義

```
{(i,j) in LINKS}
```

表のすべての × を表す（目的関数の中など）

i が定義済み

```
{("GARY",j) in LINKS}
```

GARY の行だけを取り出す＝行スライス（Supply制約）

j が定義済み

```
{(i,"LAN") in LINKS}
```

LAN の列だけを取り出す＝列スライス（Demand制約）

三つ組・より長い組への拡張

- 順序対の記法は三つ組・四つ組・任意長のリストへ自然に拡張される。
- 集合中の組はすべて同じ次元でなければならない。
- 発地・着地・製品の三つ組の部分集合 ROUTES を宣言できる。

100,000

発地10・着地100・製品100なら within {ORIG,DEST,PROD} は潜在的に最大10万要素。AMPL は宣言時にこの集合を作らず、ROUTES のデータ各組の各成分だけを検査するので効率的。

```
// 三つ組の部分集合として宣言
set ROUTES within {ORIG,DEST,PROD};

param cost {ROUTES} >= 0;
var Trans {ROUTES} >= 0;

// 三つ組の形で書く
minimize Total_Cost:
  sum {(i,j,p) in ROUTES}
     cost[i,j,p]*Trans[i,j,p];
```

```
// スライス：i と p が既定義
subject to Supply {i in ORIG, p in PROD}:
  sum {(i,j,p) in ROUTES}
     Trans[i,j,p] = supply[i,p];
```

複合集合の演算子

union / inter / diff / symdiff

第5章の単純な集合と同じ演算。次元の一致する複合集合どうしで使える

in / within / card

所属検査・部分集合検査・要素数のカウントも同様に使える

cross（デカルト積）

ORIG cross DEST は添字式 {ORIG,DEST} と同じ集合を与える

setof（反復演算子）

sum と同様に添字式＋式。定義域をモデル内で組み立てられる

setof で定義域を組み立てる／順序を条件に使う

```
// ROUTES から成分を取り出す
set ROUTES dimen 3;

set PROD = setof {(i,j,p) in ROUTES} p;
set LINKS =
   setof {(i,j,p) in ROUTES} (i,j);
```

重複する要素は無視され、新しい集合にまとめられる

```
// 条件つきの複合集合
set TRANSF =
  {i1 in ORIG, i2 in ORIG: i1 <> i2};

// PITT-PITT のような対を除外
```

順序づけられた集合と ord()

```
set ORIG ordered;
set TRANSF = {i1 in ORIG, i2 in ORIG:
   ord(i1) < ord(i2)};
```

- 「三角形」の対の集合になる — 対とその逆順の対を両方は含まない。
- ORIG で先に来る方に応じて ("PITT","CLEV") か ("CLEV","PITT") のいずれか一方だけを含む。
- 数値の集合は自然に順序づけられているので同様に扱える。

他の集合で添字づけられた集合

- 集合は個々に宣言するのが最も一般的だが、他の集合で添字づけられた集まりとしても宣言できる。
- パラメータ・変数・制約の添字づけられた集まりとほぼ同じ原理。
- 多期間生産モデルを拡張し、製品ごとに異なる市場地域を扱う例。

```
set PROD;
set AREA {PROD};
```

PROD の各要素 p について集合 AREA[p] が存在する。その要素は製品 p が販売される市場地域。

```
// 地域でも添字づけ
param market
   {p in PROD, AREA[p], 1..T} >= 0;
param revenue
   {p in PROD, AREA[p], 1..T} >= 0;

var Sell
   {p in PROD, a in AREA[p], t in 1..T}
      >= 0, <= market[p,a,t];

// 製品 p の全地域にわたる収入
sum {a in AREA[p]}
   revenue[p,a,t]*Sell[p,a,t]
```

図6-3　添字づけられた集合を使った多期間生産

```
set PROD;          # products
set AREA {PROD};   # market areas for each product
param T > 0;       # number of weeks

param market {p in PROD, AREA[p], 1..T} >= 0;
param revenue {p in PROD, AREA[p], 1..T} >= 0;

var Sell {p in PROD, a in AREA[p], t in 1..T}
               >= 0, <= market[p,a,t];

maximize Total_Profit:
   sum {p in PROD, t in 1..T}
      (sum {a in AREA[p]} revenue[p,a,t]*Sell[p,a,t]
         - prodcost[p]*Make[p,t] - invcost[p]*Inv[p,t]);

subject to Balance {p in PROD, t in 1..T}:
   Make[p,t] + Inv[p,t-1]
      = sum {a in AREA[p]} Sell[p,a,t] + Inv[p,t];
```

steelT3.mod（抜粋）

- データでは製品ごとに AREA[p] を個別に指定する。
- market・revenue は3集合で添字づけられ、製品ごとに一つの表として与えるのが便利。
- 階層関係（製品→地域）を素直に表現できる。

使い分けの指針 —— 添字づけられた集まり vs 組の集合

▽ 階層関係 → 添字づけられた集まり

- 製品 p ごとに市場地域 AREA[p] が決まるような、非対称な関係に向く。
- 「一つの製品を販売する全地域」を何度も参照するが、逆方向はほぼ使わない。

```
set orig {PROD} within ORIG;
set dest {PROD} within DEST;
set links {p in PROD} =
   orig[p] cross dest[p];

// 反復和・反復共通部分
union {p in PROD} orig[p]
inter {p in PROD} orig[p]
```

⇔ 対称関係 → 組の集合

- 発地と着地のように対等な実体どうしの関係に向く。
- 図6-5の Multi 制約のように、階層に沿わない参照が必要な場合はやや扱いにくい添字式になる。

```
// 図6-5 multic.mod（抜粋）
subject to Multi {i in ORIG, j in DEST}:
  sum {p in PROD:
     (i,j) in links[p]}
     Trans[p,i,j] <= limit[i,j];
```

✓ まとめ

1	**複合集合**	対・三つ組・より長い組を要素とする集合。組の順序が意味を持ち、次元はすべて揃える
2	**全対と部分集合**	{ORIG,DEST} は全対、within で使える対だけの部分集合 LINKS を定義できる
3	**スライス**	{(i,j) in LINKS} は、i や j がすでに定義済みの文脈では対応する行・列だけを取り出す
4	**演算子**	union・cross・setof など。cross はデカルト積、setof は反復して定義域を組み立てる
5	**添字づけられた集合**	AREA{PROD} のように他の集合で添字づけ。階層関係には添字づけの集まり、対称関係には組の集合が適する

次章：第7章「線形計画モデル― その他の話題」