AMPL SERIES · 全 22 章

AMPL：数理計画のためのモデリング言語

Σ

はじめに

数理計画法と *AMPL* モデリング言語への招待

Robert Fourer ／ David M. Gay ／ Brian W. Kernighan（日本語版）

本セクションの内容

1. **数理計画法とは何か**
   制約のもとで目的関数を最適化するという考え方の起こり
2. **数理計画法の広がり**
   線形計画法・非線形計画法・整数計画問題という 3 つの姿
3. **AMPL モデリング言語の役割**
   モデル作成者の形式をそのまま計算機に渡す代数的言語
4. **モデルとデータを分離する設計思想**
   大規模かつ保守しやすい最適化を可能にする発想
5. **本書全 22 章の構成と入手方法**
   4 部構成の全体像と AMPL ソフトウェアの試し方

数理計画法とは何か

本書の主題には二つの側面がある。第一は**数理計画法**（mathematical programming）——制約条件のもとで多変数関数を最適化することである。第二は、その開発・適用を助ける **AMPL モデリング言語** である。

「計画法（programming）」の語源

1940 年頃、大組織における活動の立案・日程計画を指す語として使われ始めた。「プランナー」は各活動の水準を**変数**として表現し、立案上の制限を等式・不等式の集まりとして数学的に記述した。

数理計画法の定式化

1. **変数**
   活動の量・水準を表す未知数
2. **制約**
   変数に関する等式・不等式の集まり
3. **目的**
   解の良し悪しを測る費用・利益などの関数
4. **最適解**
   目的を最小化／最大化する受理可能解

なぜ「目的」が鍵となる洞察か

制約を指定するだけでは複雑な業務のモデル化は難しい――制約が少なすぎれば劣った解が数多く残り、多すぎれば望ましい解が排除され、最悪の場合は解が存在しなくなる。

! 制約だけで表現すると

- 制約が少なすぎる → 劣った解が数多く条件を満たす
- 制約が多すぎる → 望ましい解が排除される
- 最悪の場合、条件を満たす解が一つも存在しなくなる

✓ 目的を追加すると

- 費用・利益など、解の良し悪しを測る目的関数を導入する
- 多くの解が制約を満たしても問題にならない
- 目的を最小化・最大化する解を一つ見つければ十分

数理計画法 = 制約のもとで目的関数を最小化・最大化すること

数理計画法の広がり

目的・制約の性質によって、数理計画問題はいくつかの特別な場合に分けられる。

L

線形計画問題

Linear Program

費用・必要量などが活動水準に厳密に比例――目的も制約もすべて線形。

数千変数でも高速・確実に解ける単体法（1940 年代後半〜）が発展の礎。

N

非線形計画問題

Nonlinear Program

目的や制約に滑らかな非線形関数が現れる場合。「大規模最適化」とも呼ばれる。

解くのはより困難だが、線形計画法の成功を受け近年になり実用計算手法が発展。

I

整数計画問題

Integer Program

一部の変数が整数値をとらなければならない場合。一般にはるかに難しい。

計算機の高速化と手法の洗練により、大規模問題も年々扱いやすくなっている。

AMPL とは何か

AMPL は数理計画法のための**代数的モデリング言語**（algebraic modeling language）である。1985 年頃に著者らによって設計・実装され、以来進化を続けてきた。

算術式が代数記法に酷似

x + y、Σ a[i,j]・x[j]、x ≥ 0 のような見慣れた数学記法をそのまま計算機で扱える

集合と添字が汎用的で強力

問題の構造に応じて柔軟に変数・制約の集まりを表現できる

数理計画特有の構造も表現

ネットワークフロー制約や区分的線形性なども言語として拡張

数式 → AMPL コード

伝統的な数学記法

```
最大化: Σj cj xj
条件:   Σj aij xj ≤ bi    （すべての i）
        xj ≥ 0
```

AMPL 記述

```
maximize Profit:
   sum {j in J} c[j] * x[j];
subject to Limit {i in I}:
   sum {j in J} a[i,j] * x[j] <= b[i];
var x {j in J} >= 0;
```

モデルとアルゴリズムをつなぐ実務の流れ

実務の数理計画法は、単にアルゴリズムを走らせて終わりではない。次のサイクルを繰り返す。



課題：モデル作成者の形式とアルゴリズムの形式の間には大きな隔たりがある

線形計画法では、制約係数行列（全制約×全変数の数表）が形式の大部分を占める。行数・列数が数百〜数十万に及ぶ疎行列を生成する「行列生成器」を書く方法は、変換作業を自動化できる一方、デバッグと保守が難しいままである。

モデリング言語という解決策

モデリング言語は、モデル作成者の形式を計算機システムへの直接の入力として表現できるように設計されている。翻訳という中間段階をすべて計算機に任せられるため、開発と保守がより経済的・信頼できるものになる。

馴染み深い表記

x+y、Σ aij xj、xj≥0 など、代数や解析を学んだ人になじみの記法をそのまま使える

多様なモデルに対応

線形・非線形・整数計画のきわめて多様なモデルに適用できる

大規模問題にも通用

数百〜数千、さらには数十万規模の変数・制約でも同じ手法が通用する

代数的モデリング言語の歩み

1950s

数理計画法の実用的なアルゴリズムが初めて使われる

1970s〜

代数的モデリング言語の開発・普及が始まる

1985 頃〜

AMPL が設計・実装され、以来進化を続けている

§ 本書の構成（全 22 章・4 部構成）

1

第 1 部　1〜4 章

線形計画法への入門

生産モデル／配合モデル／輸送・割当モデル／より大きなモデルの構築。できるだけ早く AMPL を使い始めるためのチュートリアル。

2

第 2 部　5〜8 章

AMPL の基本構成要素

単純な集合と添字／複合集合と添字／パラメータと式／変数・目的・制約。一般的なユーザーズガイドとしても使える。

3

第 3 部　9〜14 章

AMPL の高度な使い方

データの指定／データベースアクセス／モデリングコマンド／表示コマンド／コマンドスクリプト／ソルバーとのやりとり。

4

第 4 部　15〜20 章

特別な場合と一般化

ネットワーク線形計画／列単位の定式化／区分的線形計画／非線形計画／相補性問題／整数線形計画。付録 A に AMPL リファレンスマニュアル。

第 2 版と AMPL の入手方法

第 2 版について

- AMPL の中核は 10 年間で本質的に変わっておらず、第 1 版のモデルはほぼすべて現行版で動作する。
- 全面的に実質改訂したが、新規内容の大半は第 3 部に集中。
- コマンド環境の章が 1 章 → 5 章に拡張。
- データベースアクセス・スクリプト・プログラミング構文はすべて新規。

無償

www.ampl.com から「学生版」AMPL と代表的なソルバーを Windows／Unix・Linux／Mac OS X 向けに無償ダウンロード可能。数百個程度の変数・制約なら十分扱える。

`www.ampl.com/TRYAMPL`

ブラウザからダウンロードせずに AMPL を試せる

`neos.mcs.anl.gov`

NEOS サーバーのブラウザインタフェースを利用可能

★ まとめ

1	**数理計画法とは**	制約のもとで多変数の目的関数を最小化・最大化する数学的枠組み。
2	**成功の鍵は「目的」**	制約だけでなく目的関数を導入したことが、実用化を可能にした。
3	**AMPL は代数的モデリング言語**	モデル作成者の形式をそのまま計算機への直接入力にできる。
4	**モデルとデータの分離**	同じモデルを異なるデータ集合に適用でき、大規模問題にも通用する。
5	**本書は全 22 章 4 部構成**	入門チュートリアルから高度な機能・特別な場合までを体系的に解説。

次章から、*AMPL* による最初の線形計画モデル（生産モデル）を扱う。

AMPL：数理計画のためのモデリング言語

全22章シリーズ　第1章

生産モデル
利益の最大化

圧延機の生産計画から学ぶ、線形計画法とAMPLの基本

Robert Fourer, David M. Gay, Brian W. Kernighan 著
日本語版

AGENDA

本章の内容

1.1 **2変数の線形計画問題**
圧延機の生産計画を例にLPの基礎を学ぶ

1.2 **2変数の線形計画問題をAMPLで書く**
prod0.modとAMPLセッションの実行

1.3 **線形計画モデル**
代数形式による一般モデルの記述

1.4 **線形計画モデルをAMPLで書く**
集合・パラメータ・変数・目的・制約

1.5 **モデルに下限を加える**
commitパラメータによる下限制約

1.6 **モデルに資源制約を加える**
生産段階(STAGE)への拡張と限界値

1.7 **AMPLのインタフェース**
コマンド・GUI・Web/APIの選択肢

1.1 節

問題設定：圧延機とバンド・コイル

ある製鋼会社の圧延機は、鋼スラブを2種類の半製品に加工できる。

バンド（bands）とコイル（coils）
毎時の生産速度・トン当たり利益・週間の最大生産量が製品ごとに異なる。

今週の生産時間は40時間。総利益を最大にするには、バンドとコイルを何トンずつ生産すべきか？

製品	毎時トン数	トン当たり利益	最大週間トン数
バンド	200	$25	6,000
コイル	140	$30	4,000

40 時間
今週利用できる圧延機の稼働時間

1.1 節

決定変数とLPの定式化

決定変数

XB = 生産するバンドのトン数　　XC = 生産するコイルのトン数

総所要時間は40時間を超えられない：

(1/200) XB + (1/140) XC ≦ 40

生産量にはそれぞれ上限がある：

0 ≦ XB ≦ 6000　　0 ≦ XC ≦ 4000

LP 定式化

最大化:

```
25 XB + 30 XC
```

制約:

```
(1/200) XB + (1/140) XC <= 40
0 <= XB <= 6000
0 <= XC <= 4000
```

1.1 節

手計算による解法

トン当たり利益 × 毎時トン数 = 圧延機1時間当たりの利益

$5,000 /時
バンド生産の毎時利益

$4,200 /時
コイル生産の毎時利益

バンドが有利
→ 上限の6,000トンまで生産（30時間）

残り 10 時間はコイルの生産に充当 → 1,400 トン

利益 = $25 × 6,000トン + $30 × 1,400トン

$192,000

グラフによる解法（1）実行可能領域

- 横軸 XB、縦軸 XC が生産量の組合せ（解）を表す
- 垂直線：バンドの生産上限 6,000
- 水平線：コイルの生産上限 4,000
- 斜めの線：時間制約 (1/200)XB+(1/140)XC=40
- 網掛け領域＝すべての制約を満たす実行可能領域

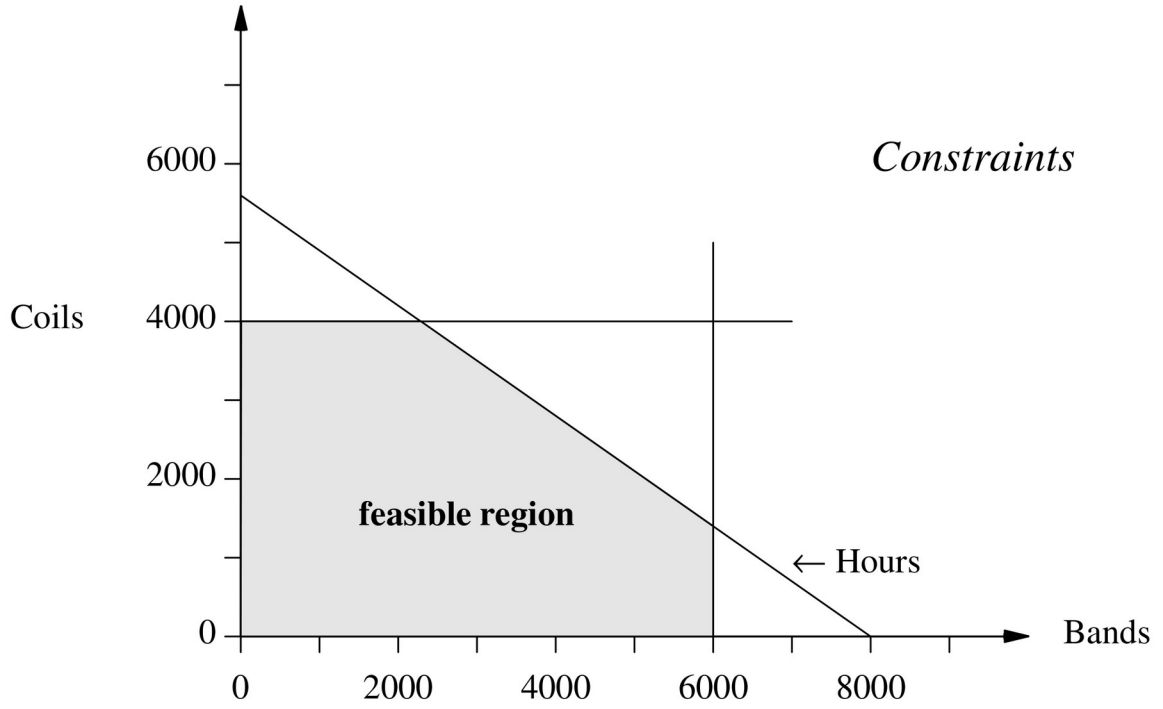


図1-1　制約条件と実行可能領域（網掛け部分）

グラフによる解法（2）等利益線

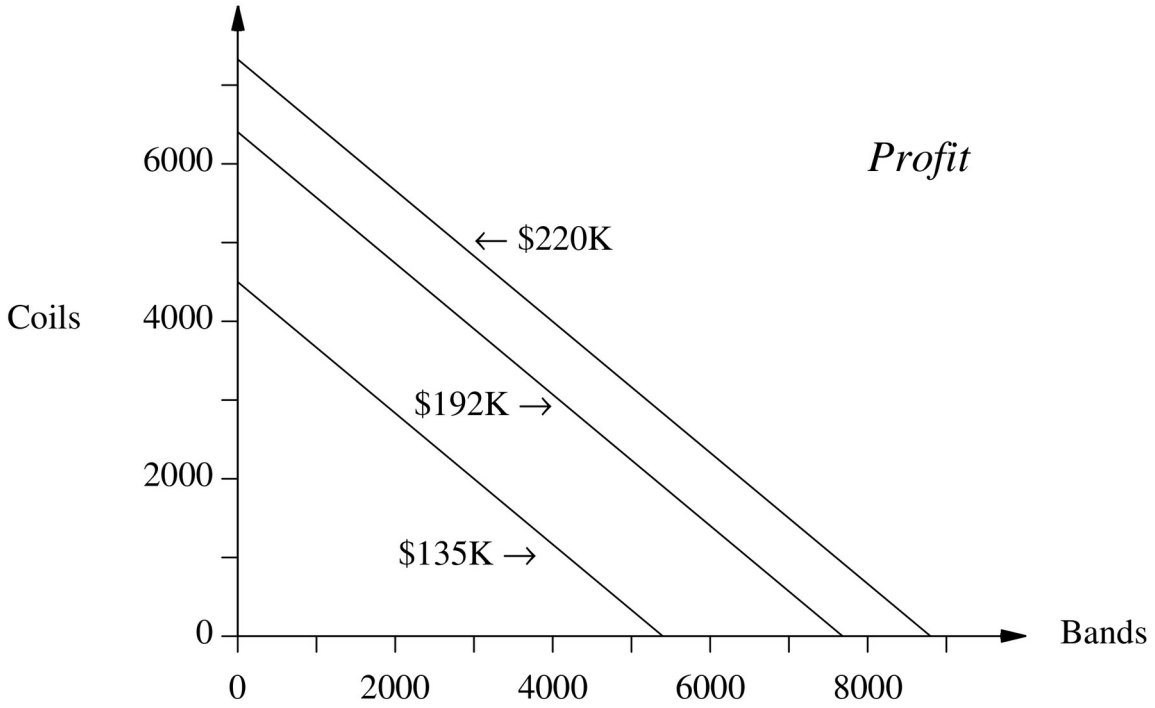


図1-2　利益（Profit）を表す等利益線

- 傾き -25/30 の平行線が、同じ利益をもたらす組合せを表す
- 例：(0, 4500) から (5400, 0) の直線 → 利益 $135,000
- 利益が高いほど、直線は右上に位置する
- 実行可能領域と交わる直線のうち、最も右上にあるものが最大利益

グラフによる解法（3）最適解

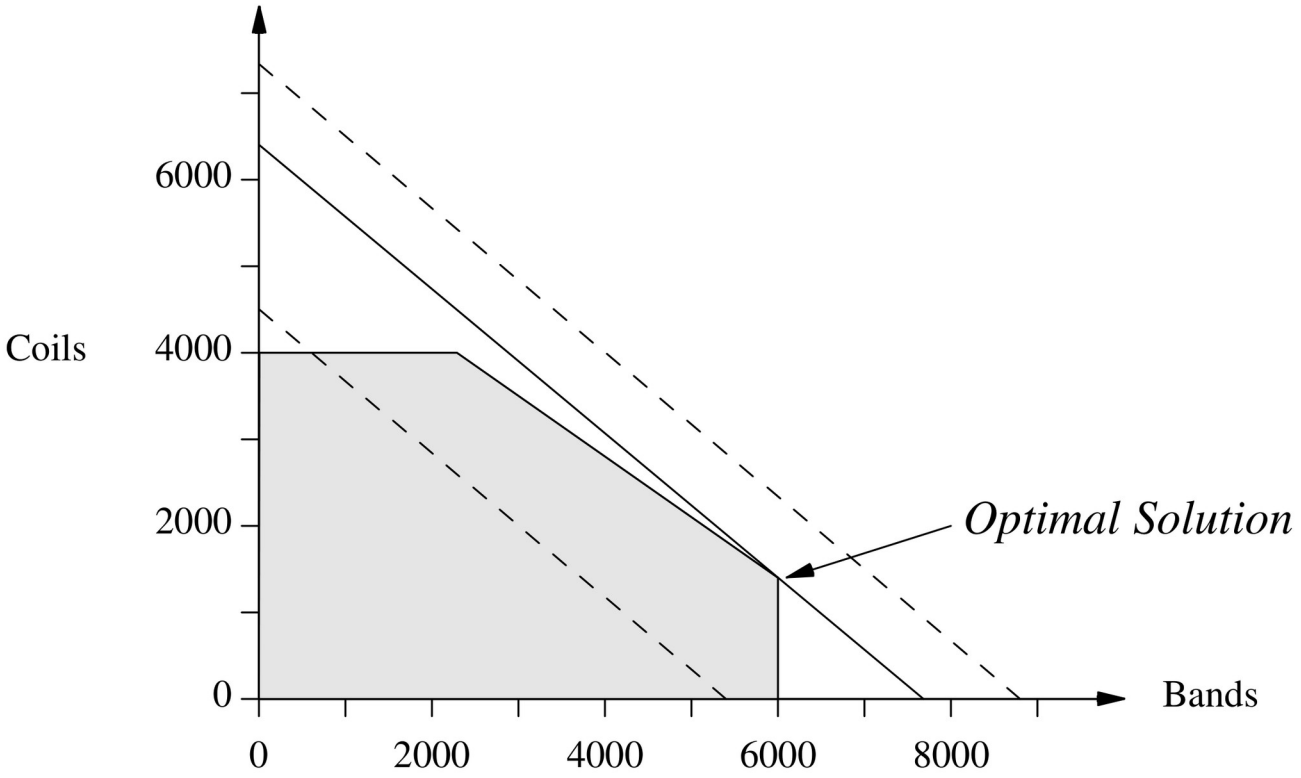


図1-3　最適解（Optimal Solution）

$220,000の等利益線は実行可能領域と交わらない。領域の隅の一点で接する直線が最適解。

6,000
XB（バンド, トン）

1,400
XC（コイル, トン）

最大利益

$192,000

1.2 節

AMPLで2変数LPを記述する

prod0.mod

```
var XB;
var XC;

maximize Profit: 25 * XB + 30 * XC;

subject to Time:
   (1/200) * XB + (1/140) * XC <= 40;
subject to B_limit: 0 <= XB <= 6000;
subject to C_limit: 0 <= XC <= 4000;
```

- var 文で各変数に名前を与える
- subject to の後に制約名（Time, B_limit, C_limit）を書く
- 掛け算には * 演算子が必須
- ≦ は <= と書く
- 代数的な記述にきわめて近い構文

1.2 節

AMPLセッションを実行する

ampl session

```
ampl: model prod0.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
2 iterations, objective 192000
ampl: display XB, XC;
XB = 6000
XC = 1400
ampl: quit;
```

model
prod0.mod を読み込む

solve
ソルバーを呼び出して解く

display
変数の最適値を表示する

quit
AMPLセッションを終了する

1.3 節

一般化：線形計画モデル（代数形式）

基本の生産モデル

```
与えられるもの:
  P = 製品の集合
  a[j] = 製品jの毎時トン数
  b    = 利用可能な時間
  c[j] = 製品jのトン当たり利益
  u[j] = 製品jの最大トン数

変数: X[j] = 製品jの生産トン数

最大化: sum j in P  c[j] * X[j]
制約:  sum j in P (1/a[j]) X[j] <= b
       0 <= X[j] <= u[j]  (各 j)
```

モデルの5つの基本構成要素

集合 (sets)
製品のような要素の集まり

パラメータ (parameters)
生産速度や利益率などの定数

変数 (variables)
ソルバーが値を決定する量

目的 (objective)
最大化・最小化すべき式

制約 (constraints)
解が満たすべき条件

AMPLモデルとデータの分離

prod.mod

```
set P;
param a {j in P};
param b;
param c {j in P};
param u {j in P};
var X {j in P};

maximize Total_Profit:
  sum {j in P} c[j] * X[j];
subject to Time:
  sum {j in P} (1/a[j]) * X[j] <= b;
subject to Limit {j in P}:
  0 <= X[j] <= u[j];
```

prod.dat

```
set P := bands coils;
param:   a    c    u :=
 bands  200  25  6000
 coils  140  30  4000 ;
param b := 40;
```

モデルとデータの分離が鍵

製品が2種類から42種類に増えても、モデル（prod.mod）は一切変更不要。データファイルを差し替えるだけで新しい問題例を解ける。



1.4 節

改良モデル：読みやすさの工夫

steel.mod

```
set PROD;                  # products
param rate {PROD} > 0;     # tons/hour
param avail >= 0;          # hours/week
param profit {PROD};       # profit/ton
param market {PROD} >= 0; # sales limit

var Make {p in PROD} >= 0,
                    <= market[p];

maximize Total_Profit:
  sum {p in PROD} profit[p]*Make[p];

subject to Time:
  sum {p in PROD} (1/rate[p])*Make[p]
     <= avail;
```

- 意味の分かる名前：PROD, rate, avail, profit, market
- 変数の上下限は var 宣言の中で直接指定
- # から行末までがコメント
- 同じ2変数LPを解いても結果は同じ（利益 $192,000）

1.5 節

モデル拡張①：生産量の下限を加える

鋼板（plate）を追加すると、コイルの生産量が0になってしまう。
現実には販売の確約があるため、生産量には下限が必要。

steel3.mod（抜粋）

```
param commit {PROD} >= 0;
var Make {p in PROD} >= commit[p],
                     <= market[p];
```

製品	commit	Make	market
bands	1000	6000	6000
coils	500	500	4000
plate	750	1028.57	3500

$194,828.57

steel3.mod の最適解の利益（bands 6000, coils 500, plate 1028.57）

1.6 節

モデル拡張②：資源制約（生産段階）を加える

生産を再加熱（reheat, 35時間）と圧延（roll, 40時間）の2段階に分割する。

steel4.mod（抜粋）

```
set STAGE;  # reheat, roll
param rate {PROD,STAGE} > 0;
param avail {STAGE} >= 0;

subject to Time {s in STAGE}:
  sum {p in PROD} (1/rate[p,s])*Make[p]
     <= avail[s];
```

$190,071
最適解の総利益

$1,800 /h
reheat の限界値

限界値（双対値）

制約を1時間緩めたときの利益改善分。reheatは+1時間ごとに$1,800、rollは+1時間ごとに$3,200の追加利益。

被約費用

coilsの下限を1トン増やすと利益は約$1.86減少。bands・plateは上下限の間にあり被約費用は実質0。

1.4 節

誤りの検出と expand コマンド

エラー例

```
steel.mod, line 8:
        syntax error
context: var Make {p in PROD}
  >>> 0 <<< <= Make[p] <=
  market[p];

steel.mod, line 11:
        make is not defined
context: maximize Total_Profit:
  sum {p in PROD} profit[p] *
  >>> make[p] <<< ;
```

expand コマンド

```
ampl: expand Total_Profit;
maximize Total_Profit:
  25*Make['bands']
  + 30*Make['coils'];

ampl: expand Time;
subject to Time:
  0.005*Make['bands']
  + 0.00714286*Make['coils']
  <= 40;
```

1.7 節

AMPLの多様なインタフェース

コマンドインタフェース

文字コマンドを打ち込む方式。全機能にアクセスでき、どの環境でも共通。本書の例で使用。

グラフィカルインタフェース

Java／Tcl-Tk／Visual Basicベースなど。ボタンやメニューで標準操作を行える。

Web・APIインタフェース

クライアント・サーバー型のアクセスや、他プログラムから呼び出すAPIも利用可能。

まとめ

1. 線形計画問題（LP）は、目的と制約を変数の線形な式で記述する。
2. 変数が2個の小規模LPなら、手計算やグラフでも最適解を求められる。
3. AMPLは代数的な記述にきわめて近い構文でLPをそのまま表現できる。
4. モデルとデータを分離することが、複雑な問題を簡潔に扱う鍵となる。
5. 下限・複数段階の資源制約なども、モデルへ自然に組み込める。

次章：第2章「食餌問題：費用の最小化」へつづく

AMPL：数理計画のためのモデリング言語

第 2 章

配合モデルとその他の投入モデル：費用の最小化

配合問題（*diet problem*）を題材に、費用を最小化する線形計画モデルの立式と *AMPL による実装を学ぶ*



AGENDA

本章の内容

2-1 配合問題の線形計画問題

夕食の価格とビタミン供給率から、明示的な LP を手で立式する

2-2 配合問題の AMPL モデル

集合・パラメータ・変数を使い、データ非依存の汎用モデルを書く

2-3 AMPL 配合モデルを使う

データを与えて解き、実行不可能性の診断や整数解も検討する

2-4 ブレンド・経済・スケジューリングへの一般化

投入・産出モデルとして書き直し、幅広い応用へつなげる

SECTION 2-1

$ 配合問題の設定：食品と栄養素

8 種類の調理済み夕食から、1 週間分（＝1 日必要量の 700% 以上）のビタミン要件を満たす最も安い組合せを選ぶ。

記号	食品	価格/パック
BEEF	ビーフ	$3.19
CHK	チキン	$2.59
FISH	魚	$2.29
HAM	ハム	$2.89
MCH	マカロニ&チーズ	$1.89
MTL	ミートローフ	$1.99
SPG	スパゲッティ	$1.99
TUR	七面鳥	$2.49

食品	A	C	B1	B2
BEEF	60	20	10	15
CHK	8	0	20	20
FISH	8	10	15	10
HAM	40	40	35	10
MCH	15	35	15	15
MTL	70	30	15	15
SPG	25	50	25	15
TUR	60	20	15	10

表：*1* パック当たりビタミン *A*・*C*・*B1*・*B2* の *1* 日必要量に対する充足率（%）

≥ 700%
各栄養素について、1 日必要量の 700% 以上を満たす

SECTION 2-1

LP の定式化：目的関数と制約

購入するパック数を X_BEEF, X_CHK, … と表す。総費用とビタミン充足量は、価格・充足率を係数とする 1 次式で書ける。

【目的関数：総費用の最小化】

```
総費用 = 3.19 X_BEEF + 2.59 X_CHK + 2.29 X_FISH + 2.89 X_HAM
        + 1.89 X_MCH + 1.99 X_MTL + 1.99 X_SPG + 2.49 X_TUR
```

【制約の例：ビタミン A の充足率 ≥ 700】

```
60 X_BEEF + 8 X_CHK + 8 X_FISH + 40 X_HAM
  + 15 X_MCH + 70 X_MTL + 25 X_SPG + 60 X_TUR  ≥  700
```

- ビタミン C・B1・B2 についても同様の式があり、いずれも 700 以上でなければならない
- 常識的な要件として、非負制約 X_BEEF ≥ 0, X_CHK ≥ 0, … , X_TUR ≥ 0 を加える

4 本
の栄養素制約（A, C, B1, B2）
＋ 8 個の非負制約

SECTION 2-1

明示的な AMPL モデル（diet0.mod）

第 1 章の生産 LP と同様に、変数・目的関数・制約をそのまま書き写せば動くモデルになる。

diet0.mod

```
var Xbeef >= 0; var Xchk >= 0; var Xfish >= 0;
var Xham >= 0; var Xmch >= 0; var Xmtl >= 0;
var Xspg >= 0; var Xtur >= 0;

minimize cost:
  3.19*Xbeef + 2.59*Xchk + 2.29*Xfish + 2.89*Xham +
  1.89*Xmch + 1.99*Xmtl + 1.99*Xspg + 2.49*Xtur;

subject to A:
  60*Xbeef + 8*Xchk + 8*Xfish + 40*Xham +
  15*Xmch + 70*Xmtl + 25*Xspg + 60*Xtur >= 700;
subject to C:
  20*Xbeef + 0*Xchk + 10*Xfish + 40*Xham +
  35*Xmch + 30*Xmtl + 50*Xspg + 20*Xtur >= 700;
subject to B1:
  10*Xbeef + 20*Xchk + 15*Xfish + 35*Xham +
  15*Xmch + 15*Xmtl + 25*Xspg + 15*Xtur >= 700;
subject to B2:
  15*Xbeef + 20*Xchk + 10*Xfish + 10*Xham +
  15*Xmch + 15*Xmtl + 15*Xspg + 10*Xtur >= 700;
```

SECTION 2-1

解いてみる：単調な最適解

amplsession

```
ampl: model diet0.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
6 iterations, objective 88.2

ampl: display Xbeef,Xchk,Xfish,Xham,
  Xmch,Xmtl,Xspg,Xtur;
Xbeef = 0
Xchk = 0
Xfish = 0
Xham = 0
Xmch = 46.6667
Xmtl = -3.69159e-18
Xspg = -4.05347e-16
Xtur = 0
```

$88.20
マカロニ&チーズ 46 2/3 パックのみを購入

- ビタミン A・B1・B2 は、ちょうど 700%（15% × 46 2/3）しか満たしていない
- ビタミン C は必要量よりずっと多く供給される
- 極小の負の値（Xmtl, Xspg）は 0 とみなしてよい
- 制約を = に変えると多様化するが、費用は $89.99 に上昇

SECTION 2-2

一般モデルへ：集合の宣言

- 受け入れられる食事にするには、線形計画問題に大幅な修正が必要
- 食品・栄養素の集合はもちろん、制約や上下限の性質も変えたくなる
- 個別のデータファイルと組み合わせられる汎用モデルに依拠すれば、こうした変更ははるかに容易になる（第 1 章の生産モデルと同じ考え方）
- このモデルは「栄養素」と「食品」という 2 つの集合を扱う

diet.mod

```
set NUTR;
set FOOD;
```

```
NUTR ： 栄養素の集合
   （例：A, C, B1, B2）

FOOD ： 食品の集合
   （例：BEEF, CHK, …）
```

SECTION 2-2

モデルが必要とする数値：パラメータ

diet.mod

```
param cost {FOOD} > 0;

param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR,FOOD} >= 0;
```

`cost{FOOD}`

各食品 1 パックの費用（正）

`f_min / f_max`

各食品を買うパック数の下限・上限

`n_min / n_max`

各栄養素の摂取量の下限・上限

`amt{NUTR,FOOD}`

amt[i,j] = 食品 j の 1 パックに含まれる栄養素 i の量

f_max の宣言では、上限が対応する下限以上でなければならないと述べるため、FOOD を走るダミー添字 j が必要。

SECTION 2-2

決定変数・目的関数・制約

diet.mod

```
var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];

subject to Diet {i in NUTR}:
   n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

- Buy[j] ：食品 j を買うパック数（決定変数）。f_min[j] と f_max[j] の間に収まる
- 目的関数 Total_Cost ：全食品 j にわたる cost[j]×Buy[j] の合計を最小化
- 制約 Diet[i] ：栄養素 i の総供給量（amt[i,j]×Buy[j] の総和）が n_min[i] と n_max[i] の「二重不等式」を満たす

SECTION 2-2

完全な配合モデル（図 2-1）

diet.mod

```
set NUTR;
set FOOD;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

param n_min {NUTR} >= 0;
param n_max {i in NUTR} >= n_min[i];

param amt {NUTR,FOOD} >= 0;

var Buy {j in FOOD} >= f_min[j], <= f_max[j];

minimize Total_Cost:  sum {j in FOOD} cost[j] * Buy[j];

subject to Diet {i in NUTR}:
   n_min[i] <= sum {j in FOOD} amt[i,j] * Buy[j] <= n_max[i];
```

SECTION 2-3

D データを与える（diet.dat）

diet.dat

```
set NUTR := A B1 B2 C ;
set FOOD := BEEF CHK FISH HAM MCH MTL SPG TUR ;

param:   cost  f_min  f_max :=
  BEEF   3.19    0     100
  CHK    2.59    0     100
  FISH   2.29    0     100
   ...    ...    ...   ...
  TUR    2.49    0     100 ;

param:   n_min  n_max :=
   A      700   10000
   ...     ...    ...  ;

param amt (tr):
         A    C   B1   B2 :=
   BEEF  60   20   10   15
   ...   ...  ...  ...  ... ;
```

amt (tr) の (tr) は
表の「転置」を表す：

列 = 第 1 添字（栄養素）
行 = 第 2 添字（食品）

- f_min / n_min はもとの要件どおり
- f_max / n_max は当面、最適解に影響しない大きな値
- モデル側を param amt {FOOD,NUTR} と変えることもできる（その場合 amt[j,i] と参照）

SECTION 2-3

モデル + データを解く／拡張の要件

amplsession

```
ampl: model diet.mod;
ampl: data diet.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
6 iterations, objective 88.2

ampl: display Buy;
Buy [*] :=
BEEF   0     MCH 46.6667
  CHK  0     MTL -1.1e-16
FISH   0     SPG -1.3e-16
  HAM  0     TUR  0
;
```

結果は先ほどと同じ（objective 88.2）。より受け入れやすい食事にするため、次の要件を加えたい：

- 多様性：各食品を 2〜10 パックの範囲で含める
- ナトリウム：総量 40,000 mg を超えない
- カロリー：総量 16,000〜24,000 の範囲
- → 新しいデータ diet2.dat に反映して再実行

SECTION 2-3

実行不可能問題を診断する

amplsession

```
ampl: model diet.mod;
ampl: data diet2.dat;
ampl: solve;
MINOS 5.5: infeasible problem.
```

- B2 の body が lb（700）ちょうど → 供給不足
- NA の body が ub（40000）に到達 → 上限が効いている
- → B2 の要求を下げるか、ナトリウム上限を緩めるかの選択

```
display Diet.lb, Diet.body, Diet.ub;
```

	lb	body	ub
A	700	1993.09	20000
B1	700	841.09	20000
B2	700	601.09	20000
C	700	1272.55	20000
CAL	16000	17222.9	24000
NA	0	40000	40000

let 文で修正

```
ampl: let n_max["NA"] := 50000;
ampl: solve;
MINOS 5.5: optimal solution found.
5 iterations, objective 118.0594032
```

SECTION 2-3

整数個のパックで最良解を求める

- 連続最適解はビーフ 5.36、スパゲッティ 9.31 パックなど端数を含む
- ビーフ 6・スパゲッティ 10 に切り上げるとナトリウム上限に違反
- ビーフ 5・スパゲッティ 9 に切り下げるとビタミン B2 が不足

$118.06 → $119.30
連続最適解 → 整数最適解（+$1.24）

dieti.mod

```
var Buy {j in FOOD} integer
   >= f_min[j], <= f_max[j];
```

amplsession

```
ampl: option solver cplex;
ampl: solve;
CPLEX: optimal integer solution;
  objective 119.3
```

- 整数性は追加の制約なので、費用が上がるのは当然
- しかし食事の内容は大きく変化：3 種類の食品の量がそれぞれ 2 パック以上変わる
- 整数計画などの「離散的」な制限は、一般に解を見つけにくくする（第 20 章で詳述）

SECTION 2-4

一般化：投入産出モデル（図 2-4）

blend.mod

```
set INPUT;   set OUTPUT;

param cost {INPUT} > 0;
param in_min/in_max {INPUT};
param out_min/out_max {OUTPUT};
param io {OUTPUT,INPUT} >= 0;

var X {j in INPUT} >= in_min[j],
                    <= in_max[j];
minimize Total_Cost:
  sum {j in INPUT} cost[j] * X[j];
```

ブレンド

原料（飼料成分・原油留分・石炭）を混合し、ナトリウムやオクタン価などの品質を満たす

経済

生産活動（投入）を組み合わせ、需要（産出）を最小費用で満たす（経済均衡と関係）

スケジューリング

勤務スケジュール（投入）で、各日の必要労働時間（産出）を満たす

前提となる仮定： 連続性（X[j] は範囲内の任意値）／ 費用の線形性／ 収量の線形性
（例：鉛の添加とオクタン価の関係のように、非線形な場合は最適化がより難しくなる）

まとめ

1. **配合問題は費用最小化 LP の典型例**

   利益最大化モデルと異なり、制約は「品質」の量に対する下限が中心になる

2. **集合とパラメータで汎用モデルを構築**

   NUTR・FOOD などの集合と param 宣言により、データを差し替えるだけで様々な問題を解ける

3. **限界値で実行不可能性を診断できる**

   Diet.lb / Diet.body / Diet.ub を調べ、どの制約が効いているかを特定し、let 文でデータを修正できる

4. **整数性は解を大きく変え、専用ソルバーが必要**

   四捨五入は容易に実行不可能になる。CPLEX など整数計画に対応したソルバーで解く

5. **ブレンド・経済・スケジューリングへ一般化**

   投入・産出モデルとして書き直せば、幅広い最小費用配分問題に応用できる

次章：等式制約とその他の *LP の話題*

3

AMPL：数理計画のためのモデリング言語

第 3 章

輸送モデルと割当モデル

発地から着地へ — 最小費用フローの基本形と、割当問題への応用

Fourer, Gay, Kernighan 著 ／ 日本語版

次 本章の内容

AGENDA

3-1 輸送問題の線形計画問題

圧延所（発地）から自動車工場（着地）への鋼コイル輸送を例に、最小費用輸送計画の問題を定式化する。

3-2 輸送問題の AMPL モデル

発地集合・着地集合を軸に、supply / demand / cost / Trans と目的関数・制約を一般化したモデルを組み立てる。

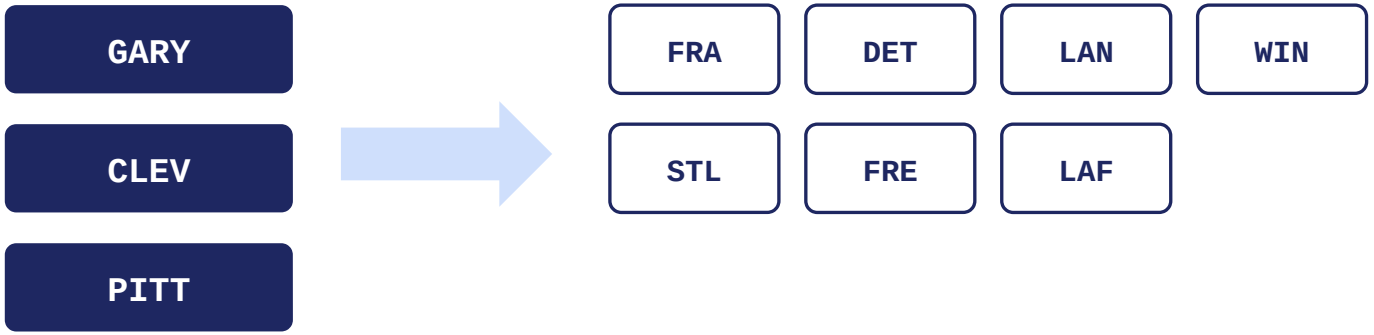
3-3 輸送モデルの他の解釈

「輸送」を伴わない対応づけ問題――割当問題――として同じモデルを再利用する方法を見る。

問 輸送問題とは

3-1　輸送問題の線形計画問題

3 か所の圧延所で生産した鋼コイルを、**7 か所の自動車工場**の注文どおりに配送する。輸送費用を最小にする配送計画は何か。



3
発地
（圧延所）

7
着地
（工場）

6,900 トン
合計輸送量
（供給＝需要）

問い：コイルを圧延所から工場へ輸送する最も安上がりな計画は何か。

GARY:FRA ＝ GARY から FRA へ輸送するトン数、というように 3 発地 × 7 着地 ＝ **21 個の変数** を用いて表す。

> 小さな輸送問題でさえ、圧延所と工場の組合せごとに 1 変数が必要になるため変数の数はすぐ増える。

表 供給地と需要地のデータ

3-1 輸送問題の線形計画問題

発地（圧延所）：週間生産量

略称	所在地	供給量（トン）
GARY	インディアナ州ゲーリー	1,400
CLEV	オハイオ州クリーブランド	2,600
PITT	ペンシルベニア州ピッツバーグ	2,900
合計		**6,900**

着地（自動車工場）：週間注文量

略称	所在地	需要量（トン）
FRA	マサチューセッツ州フレーミングハム	900
DET	ミシガン州デトロイト	1,200
LAN	ミシガン州ランシング	600
WIN	オンタリオ州ウィンザー	400
STL	ミズーリ州セントルイス	1,700
FRE	カリフォルニア州フリーモント	1,100
LAF	インディアナ州ラファイエット	1,000

供給の合計と需要の合計は一致していなければならない：

1,400 + 2,600 + 2,900 = 900 + 1,200 + 600 + 400 + 1,700 + 1,100 + 1,000 = 6,900

Σ 目的関数：総輸送費用の最小化

3-1　輸送問題の線形計画問題

1 トン当たり輸送費用（ドル）

	GARY	CLEV	PITT
FRA	39	27	24
DET	14	9	14
LAN	11	12	17
WIN	14	9	13
STL	16	26	28
FRE	82	95	99
LAF	8	17	20

目的関数（プレーンテキスト表記）

```
Minimize
  39・GARY:FRA + 27・CLEV:FRA + 24・PITT:FRA
+ 14・GARY:DET +  9・CLEV:DET + 14・PITT:DET
+ 11・GARY:LAN + 12・CLEV:LAN + 17・PITT:LAN
+ 14・GARY:WIN +  9・CLEV:WIN + 13・PITT:WIN
+ 16・GARY:STL + 26・CLEV:STL + 28・PITT:STL
+ 82・GARY:FRE + 95・CLEV:FRE + 99・PITT:FRE
+  8・GARY:LAF + 17・CLEV:LAF + 20・PITT:LAF
```

変数は全部で **21 個**（発地 3 × 着地 7）。各項は「単価 × 輸送量」で、和をとると総費用になる。

約 制約：供給制約と需要制約

3-1 輸送問題の線形計画問題

発地側の制約（圧延所ごとに 1 本、計 3 本）

```
GARY:FRA + GARY:DET + GARY:LAN + GARY:WIN
  + GARY:STL + GARY:FRE + GARY:LAF  =  1400
```

（CLEV、PITT についても同様の等式が 1 本ずつ）

着地側の制約（工場ごとに 1 本、計 7 本）

```
GARY:FRA + CLEV:FRA + PITT:FRA  =  900    (FRA 向け)
```

（他 6 工場：DET, LAN, WIN, STL, FRE, LAF についても同様）

3 + 7
= 10 本の制約

すべて非負
Trans[i,j] ≥ 0 も必要

これをすべて手入力するのは非現実的 → 一般化されたモデルが必要

集 AMPL モデル：集合とパラメータ

3-2 輸送問題の AMPL モデル

輸送モデルの根底には 2 つの基本的な集合がある：
発地（ORIG） と **着地（DEST）**。この 2 集合を軸にモデル全体を組み立てる。

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

param supply {ORIG} >= 0;   # 発地の供給量
param demand {DEST} >= 0;   # 着地の需要量

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

check 文：供給の和と需要の和が等しいかをデータ読込み後に検査し、違反していればエラーを出す。

変 AMPL モデル：変数と目的関数

3-2 輸送問題の AMPL モデル

```
param cost {ORIG,DEST} >= 0;  # 輸送単価
var Trans {ORIG,DEST} >= 0;   # 輸送量

minimize Total_Cost:
   sum {i in ORIG, j in DEST} cost[i,j] * Trans[i,j];

# 発地 i から着地 j へ Trans[i,j] 単位を、
# 単価 cost[i,j] で輸送する。総費用は
#   cost[i,j] * Trans[i,j]  を全組合せで合計。
```

同値な書き方

- sum {j in DEST, i in ORIG} cost[i,j] * Trans[i,j];
- sum {i in ORIG} sum {j in DEST} cost[i,j] * Trans[i,j];

数学的に同じ意味であれば、書き方が違っても AMPL は同一の式として扱う。

AMPL モデル：Supply / Demand 制約

3-2 輸送問題の AMPL モデル

```
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];
```

添字づけの慣習と有効範囲

- ORIG を走る添字には i を、DEST を走る添字には j を一貫して使用（AMPL の要件ではなく可読性のための慣習）。
- 添字の有効範囲（同じ意味を持つ範囲）は、その添字を定義する式の終わりまで。Demand 制約では j の有効範囲は文末のセミコロンまで、i の有効範囲は Trans[i,j] まで。
- supply（パラメータ）と Supply（制約名）は無関係 — AMPL は大文字・小文字を区別する。

解 データを与えて solve する

3-2 輸送問題の AMPL モデル

transp.dat（抜粋）

```
param: ORIG: supply :=
  GARY 1400  CLEV 2600  PITT 2900 ;

param: DEST: demand :=
  FRA 900  DET 1200  LAN 600 ...

param cost:
     FRA DET LAN ... :=
  GARY 39  14  11 ...
```

ampl セッション

```
ampl: model transp.mod;
ampl: data transp.dat;
ampl: solve;
CPLEX: optimal solution;
  objective 196200

ampl: display Trans;
  DET → GARY:0 CLEV:1200 PITT:0
  FRA → PITT:900 ...
```

196,200
最小総輸送費用（ドル）

display Trans を見ると、大半の着地は単一の圧延所から供給されているが、CLEV・GARY・PITT の 3 か所すべてが LAF へ輸送している。

較 複数の最適解が存在しうる

同じ最小費用 196,200 に対し、ソルバーによって異なる輸送計画が得られることがある（LAF 向けの内訳を比較）。

CPLEX の解	
LAF ← CLEV	400
LAF ← GARY	300
LAF ← PITT	300
目的関数値	**196,200**

SNOPT の解	
LAF ← CLEV	400
LAF ← GARY	0
LAF ← PITT	600
目的関数値	**196,200**

- このような別解の存在は輸送問題ではしばしば見られ、とりわけ目的関数の係数が切りのよい数のとき顕著。
- すべての最適解を特徴づける簡単な方法はないが、複数目的の扱い（8-3 節）でより良い解を選べる場合がある。

輸送モデルの他の解釈：割当問題

3-3 輸送モデルの他の解釈

輸送モデルは「輸送」を伴わなくてもよい。発地の量が着地と対応づけられ、目的が対応づけの価値を測るだけであれば **割当問題** になる。

例：同数の人と部屋の割当

- 発地 = 個々の人、着地 = 個々の部屋
- supply[i] = demand[j] = 1（各人は 1 部屋、各部屋は 1 人）
- Trans[i,j] = 1 なら人 i は部屋 j を使用、0 なら使用しない
- cost[i,j] = 人 i が部屋 j に付けた希望順位
- 目的：割り当てられた部屋の順位の合計を最小化 → 多くの人を満足させる割当を探す

目的関数の各項

```
cost[i,j] * Trans[i,j]

= 人 i が部屋 j に割当済み
  （Trans[i,j]=1）なら選好順位
= 未割当（Trans[i,j]=0）なら 0
```

総和 = 0 でない項の総和
= 各人の割当順位の合計

例 割当問題の例：11 人を 11 部屋へ

3-3 輸送モデルの他の解釈

assign.dat（抜粋）

```
set ORIG := Coullard Daskin Hazen ... ;
set DEST := C118 C138 C140 ... ;

param supply default 1 ;
param demand default 1 ;

param cost:  C118 C138 ... :=
  Coullard  6   9  ...
```

結果（一部）

人	割当部屋	希望順位
Coullard	C118	1
Iravani	C138	2
Hazen	C246	1
Hopp	D237	1
Nelson	C140	4
… 他 6 名		

28
目的関数値
（希望順位の合計）

0 と 1
のみ Trans[i,j] が取る値

supply・demand・上下限がすべて整数 → 最適解は自動的に整数値になる（輸送モデルの特別な性質）

注意：この整数性は輸送モデル特有の性質であり、他の問題では一般に保証されない（整数計画は第 20 章）。

要 まとめ

1. 輸送問題は、発地の供給と着地の需要を最小費用で結ぶ最小費用フロー問題の最も基本的な形である。

2. AMPL モデルは 2 つの集合（ORIG, DEST）を軸に、supply / demand / cost / Trans と目的関数・Supply/Demand 制約だけで一般化できる。

3. sum{} を使えば目的関数・制約を簡潔に書け、データを差し替えるだけで規模の異なる問題にも対応できる。

4. 同じ最小費用でも複数の最適解が存在しうる（ソルバーにより輸送計画が異なる場合がある）。

5. 割当問題（supply=demand=1）は輸送モデルの特殊ケースで、整数の供給・需要のもとでは最適解が自動的に整数になる。

次章：第 4 章 へ続く

全22章シリーズ ・ 第4章

AMPL: 数理計画のためのモデリング言語

より大きなモデルの構築

多品種輸送・多期間生産・生産と輸送の統合 — 小さな*LP*を結び合わせて大きなモデルを組み立てる

4.1 多品種輸送

4.2 多期間生産

4.3 生産と輸送

原著: Robert Fourer, David M. Gay, Brian W. Kernighan

AGENDA

本章の内容

4.1 多品種輸送モデル

前章の輸送モデルを拡張し、複数の製品を同時に発地から着地へ輸送する。発地・着地間の総輸送量を制限するMulti制約も導入する。

4.2 多期間生産モデル

生産計画を1週間ではなくT週間に拡張する。在庫変数Invを使い、Balance制約で各週の生産・販売・在庫をつなぐ。

4.3 生産と輸送のモデル

圧延所ごとの生産モデルと製品ごとの輸送モデルを1つに統合し、生産と輸送の両方の費用を最小化する。

INTRODUCTION

大きな線形計画問題はどう生まれるか

経路A：個々のモデルを精緻化する

添字集合を細かく分ければLPは大きくなる。だがある点を超えるとデータ維持も解の理解も難しくなり、細かくする利益は薄れる。

経路B：小さなモデルを結び合わせる

（本章のテーマ）

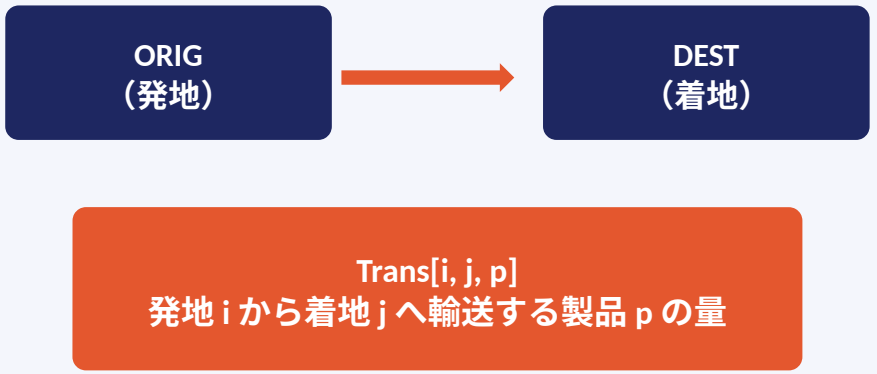
同種の単純なLPが多数生まれる場面で、それらを結ぶ変数や制約を加えると、全体は1つの大きなLPになる。

本章で扱う3つの結合パターン

- 製品ごとの輸送問題
- 週ごとの生産問題
- 生産×輸送の組合せ

製品の集合 PROD を追加する

- 前章の輸送モデルは単一商品のみを扱っていた
- 新しい集合 PROD（製品）を定義し、supply・demand・cost・Trans すべての添字に PROD を加える
- 発地 i から輸送される製品 p の量：supply[i, p]
- i から j へ輸送される製品 p の量：Trans[i, j, p]
- check 文も PROD で添字づけ、製品ごとに「供給の合計 = 需要の合計」を検証する



AMPLモデル：multi.mod（図4-1）

前半

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

param supply {ORIG,PROD} >= 0;
param demand {DEST,PROD} >= 0;

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

param limit {ORIG,DEST} >= 0;

param cost {ORIG,DEST,PROD} >= 0;
var Trans {ORIG,DEST,PROD} >= 0;
```

後半

```
minimize Total_Cost:
   sum {i in ORIG, j in DEST, p in PROD}
      cost[i,j,p] * Trans[i,j,p];

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

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

subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

Multi制約：輸送能力を共有する

- 発地・着地の組合せで添字づけたパラメータ limit[i, j] を追加する
- Multi制約：各 (i, j) について、全製品 p の輸送量の合計が limit[i, j] を超えない
- この制約があると、ある製品の輸送量は他の製品の輸送量に左右されるため、製品ごとに独立して解くことはできなくなる

AMPL

```
param limit {ORIG,DEST} >= 0;

subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

規模の例

発地 5 ・ 着地 20 ・ 製品 10 のとき

1,000
変数の数
（発地×着地×製品）

250
制約の数
（発地+着地）×製品

見かけほど大きくない

limit制約がなければ、製品の輸送は互いに独立なので、製品ごとに小さな輸送問題を別々に解く方が効率的である。limit制約が加わって初めて、1つの大きなLPを解く必要が生じる。

3次元データを2次元の表に切り分けて表示

ampl

```
ampl: model multi.mod; data multi.dat; solve;
CPLEX 8.0.0: optimal solution; objective 199500

ampl: display {p in PROD}: {i in ORIG, j in DEST} Trans[i,j,p];
Trans[i,j,'bands'] [*,*] (tr)
:    CLEV GARY PITT   :=
DET     0     0   300
FRA   225     0    75
FRE     0     0   225
LAF   225     0    25
LAN     0     0   100
STL   250   400     0
WIN     0     0    75
;
```

切り分け表示の考え方

ORIG × DEST × PROD の3次元データは、1つの添字（ここでは製品）に沿って切り分け、2次元の表の集まりとして表示する。display コマンドは切り分ける添字を自動推測するが、明示的な添字式を書けば表示方法を指定できる。

端数輸送の悩み

PITT→LAFへのバンド輸送はわずか25トン。「0か50トン以上」という条件は線形制約では表せない ―― 第20章の整数計画法で扱う。

coils・plate の表も同様に表示される（紙面の都合により省略）

週の集合 1..T を追加する

- 図1-4aの生産モデルを、1週間ではなくT週間分の計画に拡張する
- 1からTまでの整数の集合を表す略記 1..T を使う
- rate 以外の全パラメータ・変数をこの集合上に複製する
- 週ごとに制約 Time ができ、目的関数は製品と週の両方について合計する

図4-3

```
set PROD;                          # products
param T > 0;                       # number of weeks

param rate {PROD} > 0;             # tons/hour produced
param avail {1..T} >= 0;           # hours available
param profit {PROD,1..T};          # profit per ton
param market {PROD,1..T} >= 0; # limit on tons sold

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

maximize Total_Profit:
   sum {p in PROD, t in 1..T}
      profit[p,t] * Make[p,t];

subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p])
      * Make[p,t] <= avail[t];
```

在庫で週をつなぐ：Make・Inv・Sell

- Make[p, t] は生産量のみを表し、必ずしも販売量とは一致しなくなる
- 新しい変数 Inv[p, t]（在庫）と Sell[p, t]（販売量、上限 market[p,t]）を追加する
- 初期在庫パラメータ inv0[p] を、制約 Init_Inv によって Inv[p, 0] に固定する

Balance制約（週をつなぐ鍵）

生産 Make[p,t] + **前週末の在庫 Inv[p,t-1]** = **販売 Sell[p,t]** + **今週末の在庫 Inv[p,t]**

意味：生産量+繰り越された在庫 = 販売量+繰り越す在庫（第1週では Inv[p,0] が初期在庫 inv0[p] になる）

解の解釈：在庫を使って高値の週に売る

Make（生産量, トン）

週	bands	coils
1	5,990	1,407
2	6,000	1,400
3	1,400	3,500
4	2,000	4,200

Inv（在庫量, トン）

週	bands	coils
0	10	0
1	0	1,100
2	0	0
3	0	0
4	0	0

Sell（販売量, トン）

週	bands	coils
1	6,000	307
2	6,000	2,500
3	1,400	3,500
4	2,000	4,200

第1週に生産したコイル1,100トンは、より高値で売れる第2週まで在庫として持ち越されている。第2〜4週はコイルが上限まで販売され、バンドが残りの生産能力を埋めている。

生産モデルと輸送モデルを統合する

生産モデル
（圧延所 ORIG ごとに複製）
Make[i,p] ・ 制約 Time

輸送モデル
（製品 PROD ごとに複製）
Trans[i,j,p] ・ 制約 Supply/Demand

```
Make[i, p]  =  supply[i, p]
```

- ORIG（発地＝圧延所）と PROD（製品）の集合は両モデルで共通
- 輸送モデルの supply[i,p]（発地で利用可能な量）は、生産モデルの Make[i,p]（発地で生産される量）と実は同じもの
- supply の定義を落とし、Supply制約を「Σ(j∈DEST) Trans[i,j,p] = Make[i,p]」に書き換えて2つのモデルを1つに統合する

AMPLモデル：steelP.mod（図4-6）

前半

```
set ORIG;   # origins (mills)
set DEST;   # destinations
set PROD;   # products

param rate {ORIG,PROD} > 0;
param avail {ORIG} >= 0;
param demand {DEST,PROD} >= 0;

param make_cost {ORIG,PROD} >= 0;
param trans_cost {ORIG,DEST,PROD}
   >= 0;

var Make {ORIG,PROD} >= 0;
var Trans {ORIG,DEST,PROD} >= 0;
```

後半

```
minimize Total_Cost:
   sum {i in ORIG, p in PROD}
      make_cost[i,p] * Make[i,p] +
   sum {i in ORIG, j in DEST, p in PROD}
      trans_cost[i,j,p] * Trans[i,j,p];

subject to Time {i in ORIG}:
   sum {p in PROD} (1/rate[i,p])
      * Make[i,p] <= avail[i];

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

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

解の解釈：費用の内訳と双対値

1,392,175
総費用（objective）

1,215,250
生産費用（約87%）

176,925
輸送費用（約13%）

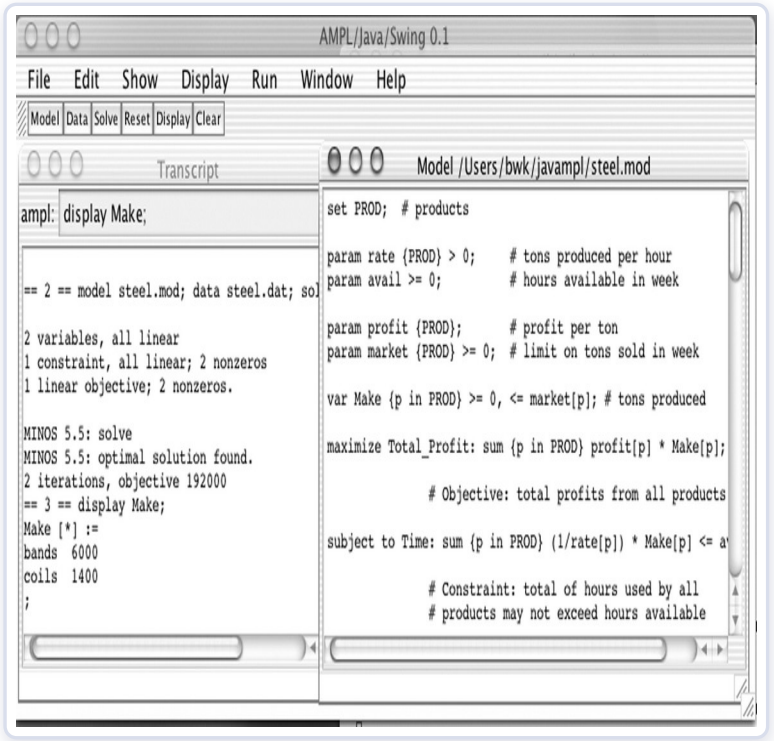
- 生産費用が総費用の大半を占め、輸送費用は相対的に小さい
- AMPLは任意の正しい式（合計など）を評価・表示できる —— display sum{...} で費用内訳を直接確認できる

Time制約の双対値（限界価値）：GARY -2800 ／ CLEV -1300 ／ PITT 0
能力を追加するならGARYへの投資が総費用に最も大きく効き、PITTに追加しても効果はない。

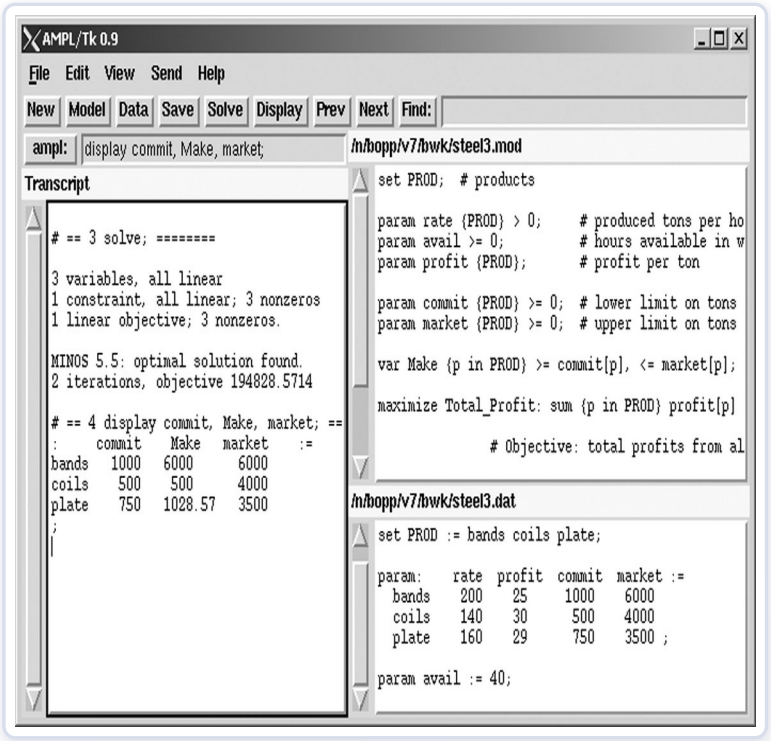
REFERENCE

AMPLの対話的モデリング環境（画面例）

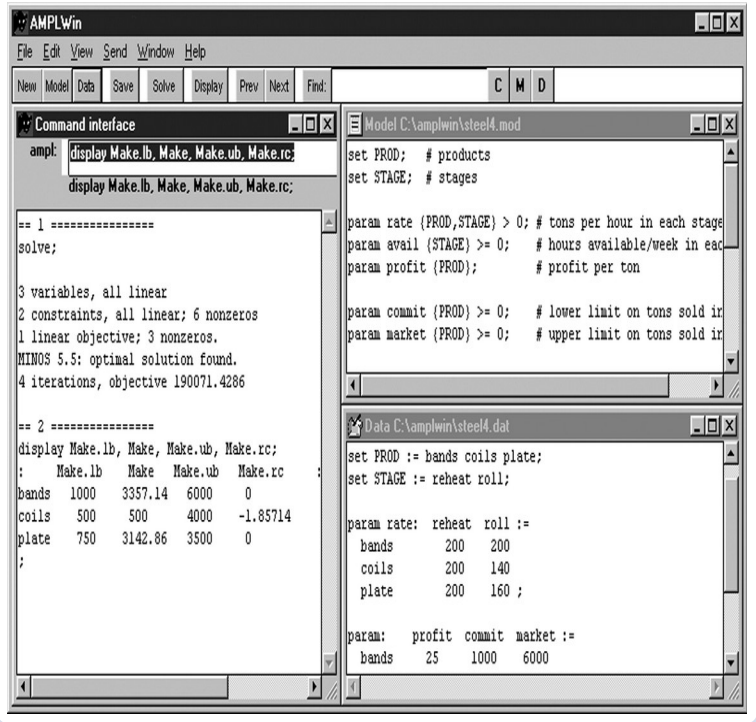
さまざまなAMPL統合環境（IDE）で、モデルの編集・solveの実行・display結果の確認を対話的に行っている様子



AMPL/Java/Swing — モデルの編集・solveの実行・displayによる結果確認という基本操作の例



AMPL/Tk — 変数に上限・下限（commit / market）を持たせたモデルの実行例



AMPLWin — 複数の添字集合（PROD × STAGE）でパラメータを添字づけたモデルの実行例

SUMMARY

本章のまとめ

- 大きなLPの多くは、小さな単純なLPを結び合わせることで生まれる
- 多品種輸送モデルはPROD集合を加えて添字を拡張し、Multi制約で発地・着地間の総輸送量を制限できる
- 多期間生産モデルは 1..T 集合で週を複製し、Balance制約（生産＋繰越在庫＝販売＋繰越在庫）で週をつなぐ
- 生産と輸送のモデルは、Make[i,p] = supply[i,p] という対応に着目して2つのモデルを1つに統合する
- AMPLは集合・添字を活用することで、モデルの本質的な論理を保ったまま規模を拡張できる

AMPL：数理計画のためのモデリング言語

Σ

第5章

単純な集合と添字

集合はモデルの骨格――順序のない集合・数値の集合・集合演算・添字式・順序集合を学ぶ

本章の内容

第5章 単純な集合と添字 —アジェンダ

5-1	**順序のない集合**	文字列の集合を宣言・列挙する基本形
5-2	**数値の集合**	数値リテラルと ".." "by" による範囲指定
5-3	**集合演算**	union / inter / diff / symdiff で新しい集合を作る
5-4	**所属に関する演算と関数**	in / within と card() で所属・部分集合を調べる
5-5	**添字式**	{i in S} の形でモデル構成要素や総和を添字づける
5-6	**順序集合**	ordered / circular と prev・next・ord などの関数

導入

集合はAMPLモデルの最も基本的な構成要素

- パラメータ・変数・制約のほとんどすべては集合で添字づけられる
- 多くの式は集合上の演算（総和など）を含む
- 集合による添字づけこそ、簡潔なモデルで大規模な数理計画問題を記述可能にする機能
- AMPL は多種多様な集合の型と演算を提供する

集合の要素は…

- **文字列 or 数値**

  "coils" のような文字列、あるいは 1990 のような数値

- **順序あり or なし**

  既定は無順序。ordered / circular で順序を持たせられる

- **単独 or 組（タプル）**

  順序対・三つ組など、より長い組としても現れる（第6章）

集合の宣言と文字列リテラル

- 最も初歩的な集合＝文字列の順序のない集まり
- キーワード set と名前だけで宣言できる
- 名前は数字で始まらない文字・数字・下線の並び
- 大文字と小文字は区別される（"fish"≠"Fish"≠"FISH"）
- 文字列は単一引用符または二重引用符で区切る

```
set PROD;

# 予約語も再定義できる例
set prod;
```

引用の例

'A&P'
　単一引用符で区切る文字列
"Bell+Howell"
　二重引用符で区切る文字列

リテラルな集合とモデル／データの分離

```
set PROD = {"bands", "coils", "plate"};

{}                  # 空集合
{"bands"}           # 要素数 1 の集合
```

✓

データ分離が基本

要素は通常、モデルではなくデータ（第9章）で指定するのが大半の応用で推奨される

✓

リテラルが有効な場面

要素数が少ない／モデルの基本要素／頻繁に変わらない場合に限って直接列挙する

✓

"bands" と {"bands"} は別物

文字列そのものと、要素が1個の集合は AMPL 上で区別される

数値リテラルと ".." "by" による範囲指定

- 集合の要素は数値でもよい（文字列との混在も可能）
- {1,2,3,4,5,6} は 1..6 とより簡潔に書ける
- 既定の間隔は 1。by 句で任意の間隔を指定できる
- 数値要素は算術式でそのまま使える（例：Inv[p,t-1]）

```
set YEARS = 1990 .. 2020 by 5;

# 展開すると
{1990,1995,2000,2005,
 2010,2015,2020}
```

より良い書き方

```
param start integer;
param end > start integer;
param interval > 0 integer;
set YEARS = start..end by interval;
```

± 浮動小数点の丸め誤差に注意

集合の要素は、実数の場合は最も近い表現可能な浮動小数点数として格納される。
0 や 1 がちょうど現れると思うかもしれないが、丸め誤差のためそうはならない。

```
ampl: display -5/3 .. 5/3 by 1/3;
set -5/3 .. 5/3 by 1/3 :=
-1.6666666666666667   0.33333333333333326
-1.3333333333333335   0.6666666666666663
-1                    0.9999999999999998
-0.6666666666666667   1.3333333333333333
-0.3333333333333335   1.6666666666666663
-2.220446049250313e-16;
```

- 正確な値に依拠するモデルでは、集合に分数を使うのは避けるのが賢明
- 整数要素なら通常は問題ない（IEEE 標準では約 10^16 まで正確に表現可能）

5-3　集合演算

既存の集合から新しい集合を作る4つの演算子

`A union B`	**和**：A か B のいずれかに属する
`A inter B`	**共通部分**：A と B の両方に属する
`A diff B`	**差**：A に属し B に属さない
`A symdiff B`	**対称差**：A か B に属するが両方には属さない

union / diff / symdiff は同順位、inter はそれより優先度が高い（左結合）

```
ampl: set Y1 = 1990..2020 by 5;
ampl: set Y2 = 2000..2025 by 5;
ampl: display Y1 union Y2,
                Y1 inter Y2;
Y1 union Y2 := 1990 1995 2000
   2005 2010 2015 2020 2025;
Y1 inter Y2 := 2000 2005 2010
   2015 2020;
```

配合モデルでの活用例（図5-1 dietu.mod）

- 下限のある栄養素 MINREQ、上限のある栄養素 MAXREQ を宣言
- 全栄養素 NUTR を毎回別に指定させると、整合性の保証がモデル利用者任せになる
- 代わりに NUTR を union で自動的に定義すれば、冗長性や誤りを防げる
- 原則：データで与えるのは必要最小限の集合、他は式で定義する

```
set MINREQ;  # 下限がある栄養素
set MAXREQ;  # 上限がある栄養素

set NUTR = MINREQ union MAXREQ;

param n_min {MINREQ} >= 0;
param n_max {MAXREQ} >= 0;
param amt {NUTR,FOOD} >= 0;
```

in / within / not と card()

in	`"B2" in NUTR`	要素が集合に属するとき真（代数記法の ∈ に対応）
within	`MINREQ within NUTR`	左の集合がすべて右の集合に属するとき真（⊆ に対応）
not in / not within	`j not in FOOD`	結果の真偽を反転する
card( )	`card(NUTR)`	集合の要素数（濃度）を返す組込み関数

宣言に直接 within を使うと部分集合であることをAMPLに強制できる： **`set MAXREQ within NUTR;`**

波括弧 { } で書く、AMPLの添字づけの基本形

- 代数記法の「すべての $i \in P$ について」に対応するのが添字式 {...}
- 最も単純な形は、波括弧に囲まれた集合名または集合式
- t や j のような名前は「ダミー添字」と呼ばれる
- パラメータ・変数の宣言は param・var に続けて全く同じ書き方ができる

```
param rate {PROD} > 0;
param avail {1..T} >= 0;

param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

var Buy {j in FOOD}
   >= f_min[j], <= f_max[j];
```

ダミー添字の有効範囲と条件付き添字式

- ダミー添字が有効なのは、それを定義する添字式の有効範囲の内側だけ
- 有効範囲が終われば未定義に戻るので、同じ添字名を繰り返し使ってよい
- 集合名の後に「: 条件」を続けると、条件を満たす要素だけの部分集合になる
- 空集合で添字づけても誤りではない（該当する構成要素は生成が飛ばされる）

```
{j in FOOD: f_max[j] - f_min[j] < 1}

set NUTREQ =
   MAXREQ union {i in MINREQ: n_min[i] > 0};
```

```
subject to Food_Ratio {j in FOOD}:
   Buy[j] >= min_frac[j] *
      sum {jj in FOOD} Buy[jj];

# 外側の j と区別するため jj を使用
```

display 文での添字式の活用（図5-1・5-2に基づく）

```
ampl: display MAXREQ union {i in MINREQ:
                              n_min[i] > 0};
set ... := A NA CAL C;

ampl: display {j in FOOD: Buy[j] > f_min[j]};
set ... := CHK MTL SPG;

ampl: display {i in MINREQ:
                Diet_Min[i].slack = 0};
set ... := C CAL;
```

対話コマンドの添字式では変数・制約も条件に参照できるが、モデル内で参照できるのは集合・パラメータ・ダミー添字だけ。

ordered / circular と前後を扱う関数

- ordered / circular を加えると、要素の並び順をAMPLが保持する
- ordered：最初の要素に前者なし、最後の要素に後者なし
- circular：最後の次は最初、最初の前は最後とみなす

- **first(S) / last(S)**　絶対位置の要素
- **prev(t) / next(t)**　相対位置の要素
- **ord(t) / ord0(t)**　集合内での位置（番号）

```
set WEEKS ordered;

subject to Balance
  {p in PROD, t in WEEKS: ord(t)>1}:
  Make[p,t] + Inv[p,prev(t)]
   = Sell[p,t] + Inv[p,t];

subject to Balance0 {p in PROD}:
  Make[p,first(WEEKS)] + inv0[p]
   = Sell[p,first(WEEKS)]
     + Inv[p,first(WEEKS)];
```

5-6　順序集合

区間式と定義済みの無限集合

`interval [a,b]`	a ≤ x ≤ b の実数区間
`interval (a,b]`	a < x ≤ b の実数区間
`integer [a,b]`	a ≤ x ≤ b の整数区間

param rate {PROD} in (0,maxrate); は *param rate {PROD} > 0, <= maxrate;* と同じ意味

```
定義済みの無限集合
Reals        # すべての浮動小数点数
Integers     # すべての整数
ASCII        # ASCII 照合順序
EBCDIC       # EBCDIC 照合順序
Display      # display の並び順

set PROD ordered by ASCII;
```

✓ 第5章のまとめ

- **集合はAMPLモデルの基本構成要素**
  パラメータ・変数・制約のほとんどは集合で添字づけられる
- **順序のない集合と数値の集合**
  set宣言、文字列・数値のリテラル、".." "by" 記法
- **集合演算子は4種類**
  union（和）・inter（共通部分）・diff（差）・symdiff（対称差）
- **所属演算と添字式が核心**
  in / within / card() と {i in S: 条件} がモデル全体を貫く
- **順序集合で「前後」を扱う**
  ordered / circular、first・last・prev・next・ord 関数

次章（第6章）：対や三つ組の集合、集合の添字づけられた集まりを含む複合集合へ

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」



AMPL：数理計画のためのモデリング言語

第 7 章　パラメータと式

宣言・演算子・条件式でモデルの数値を組み立てる

本章の内容

7-1 パラメータの宣言

7-2 算術式

7-3 論理式と条件式

7-4 パラメータへの制限

7-5 計算されるパラメータ

7-6 ランダムに生成されるパラメータ

7-7 論理パラメータ

7-8 記号パラメータ

スカラーと添字づけられたパラメータ

- 最も単純な宣言はキーワード param と名前だけ：param T; と書けば以降どこでも T で数値を参照できる。
- 名前の後に添字式を続けると、集合の各要素につき一つのパラメータが定義される。
- 参照するときは名前に角括弧で「添字」を添える：avail[i]、demand[j,p] のように。
- 添字は、もとの添字集合の要素に評価される限り、任意の式でよい。

param 宣言の例

```
param T;

param avail {1..T};
param demand {DEST,PROD};
param revenue
   {p in PROD, AREA[p], 1..T};
```

添字を付けて参照する

```
avail[i]
demand[j,p]
revenue[p,a,t]
```

ベクトル・行列・疎行列としてのパラメータ

ベクトル

整数列で添字づけ

```
param avail {1..T};

avail[1], avail[2],
..., avail[T]
```

整数列にわたる添字づけは配列・ベクトルに対応するが、文字列の集合で添字づけてもよい。

行列

対の全組合せで添字づけ

```
set ORIG;  set DEST;
param cost {ORIG,DEST};

cost[i,j]  (すべての i,j)
```

ORIG の i と DEST の j のあらゆる組合せについて cost[i,j] が存在する。

疎な行列

対の部分集合で添字づけ

```
set LINKS within {ORIG,DEST};
param cost {LINKS};

cost[i,j]  ((i,j) in LINKS)
```

(i,j) が LINKS の要素であるときのみ cost[i,j] が存在する「疎な」行列となる。

7-2 算術式

演算子と優先順位（表 7-1）

通常の書き方	代替の書き方	型（被演算子 → 結果）
`if-then-else`		論理,算術 → 算術
`or`	`||`	論理 → 論理
`exists forall`		論理 → 論理
`and`	`&&`	論理 → 論理
`not`（単項）	`!`	論理 → 論理
`< <= = <> > >=`	`< <= == != > >=`	算術 → 論理
`in  not in`		対象,集合 → 論理
`+  -  less`		算術 → 算術
`sum prod min max`		算術 → 算術
`*  /  div  mod`		算術 → 算術
`+ -`（単項）		算術 → 算術
`^`	`**`	算術 → 算術

優先順位が下ほど高い（表は下に行くほど強く結合）。

べき乗と if-then-else は右結合、他は左結合。

if-then-else の論理被演算子は if の後、算術被演算子は then / else の後に置く。

組込み関数と反復演算子

表 7-2　組込み算術関数（抜粋）

abs(x) 絶対値	**sqrt(x)** 平方根	**exp(x)** 指数関数
log(x) 自然対数	**log10(x)** 常用対数	**min/max(...)** 最小・最大
sin cos tan 三角関数	**asin acos atan** 逆三角関数	**sinh cosh tanh** 双曲線関数

min・*max* 以外は再定義可能（元の意味は使えなくなる）。
card・*ord* も算術値を返す。

反復総和 sum

```
sum {i in ORIG} supply[i]

sum {i in ORIG, j in DEST}
   cost[i,j] * Trans[i,j]
```

sum の優先順位に注意

```
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] - ...);
```

- 他の反復演算子：prod（積）・min・max。sum は * より優先順位が低く、+ - より高い。

7-3 論理式と条件式

比較・論理演算子

比較演算子

=	等しい	<>	等しくない
<	より小さい	<=	以下
>	より大きい	>=	以上

- and：両方真のとき真 / or：少なくとも一方が真のとき真 / not：真偽を反転。
- 反復版：exists（反復 or）、forall（反復 and）。
- 優先順位：比較・in・算術 > not > and > or。

論理式の例

```
T >= 0 and T <= 10

i in MAXREQ or n_min[i] > 0

exists {i in ORIG} demand[i] > 10
forall {i in ORIG} demand[i] > 10
```

not の優先順位（表 7-1 参照）

```
not i in MAXREQ or
   n_min[i] > 0 and n_min[i] <= 10
# は次のように解釈される：
(not (i in MAXREQ)) or
  ((n_min[i]>0) and (n_min[i]<=10))
```

if-then-else 条件演算子

`if a then b else c`

- a は論理式。真なら値 b、偽なら値 c をとる。
- c が 0 のとき else c は省略できる。
- b と c は同じ種類の式であれば算術式・文字列式・集合式でもよい。
- then / else は他のどの演算子より優先順位が低いので、文末以外では括弧が必要。

二つの制約を一つにまとめる例

```
subject to Balance0 {p in PROD}:
  Make[p,first(WEEKS)] + inv0[p]
  = Sell[p,first(WEEKS)]
     + Inv[p,first(WEEKS)];

subject to Balance
   {p in PROD, t in WEEKS: ord(t)>1}:
  Make[p,t] + Inv[p,prev(t)]
     = Sell[p,t] + Inv[p,t];

# ↓ if-then-else でひとつに統合
subject to Balance {p in PROD, t in WEEKS}:
  Make[p,t] +
   (if t=first(WEEKS) then inv0[p]
    else Inv[p,prev(t)])
   = Sell[p,t] + Inv[p,t];
```

宣言時に制限句でデータを検査する

- integer・binary、または比較演算子＋算術式を宣言に付けられる。
- 添字づけられた宣言では制限が各要素に適用される。ダミー添字を使えば要素ごとに異なる制限も可能。
- 同じ宣言中のパラメータ自身を参照する自己参照には if-then-else が必要（t-1 が未定義になる場合を避ける）。
- 一つの宣言に複数の制限句を（任意でコンマ区切りで）置ける。

整数かつ 1 より大きい

```
param T > 1 integer;

param f_max {j in FOOD} >= f_min[j];
```

対話セッションでのエラー例

```
error processing param T:
    failed check: param T = 1
        is not > 1;
```

自己参照には if-then-else

```
param cumulative_market {p in PROD, t in 1..T}
  >= if t>1 then cumulative_market[p,t-1];
```

7-4 パラメータへの制限

check 文によるデータ検証

- check 文は「データのみに対する制約」。制限句より複雑な条件を課せる。
- 制限節は論理式で、既定の集合・パラメータ・添字式のダミー添字を使える。
- データ読込み後に真でなければならず、失敗するとデータをすべて拒絶する。
- 制限句・check 文の利用は強く推奨：誤ったデータによる誤ったLPの生成を防げる。

単純な check（供給＝需要）

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

添字づけられた check

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

早期発見が鍵：捕らえられなかった誤りは、最悪の場合もっともらしい*誤った解を生み、検出されないまま残る。*

7-5 計算されるパラメータ

= による代入句で新しいパラメータを定義

- 代入句は = 演算子を使い、パラメータをある式に等しく設定する（制限句は不等式だけ）。
- 複雑な式を目的・制約に直接書くより、計算されるパラメータを介した方がモデルは読みやすい。
- = で定義したパラメータには、データ文で値を指定できない（エラーになる）。
- セッション中、= による定義はすべて最新に保たれる（依存元を変えれば自動的に再計算）。

需要シェアの正規化

```
param share {DEST} >= 0, <= 100;
param tot_sh =
   sum {j in DEST} share[j];

param tot_dem {PROD} >= 0;
param demand {j in DEST, p in PROD}
   = share[j]*tot_dem[p]/tot_sh;
```

最小在庫を計算する

```
param frac > 0;
param market {PROD,1..T+1} >= 0;
param mininv {p in PROD, t in 0..T}
   = frac * market[p,t+1];
var Inv {p in PROD, t in 0..T}
   >= mininv[p,t];
```

D

7-5 計算されるパラメータ

default 演算子と再帰的定義

=（代入句）

パラメータを定義する。データ文で値を指定できない。依存元が変われば常に再計算される。

default（既定句）

パラメータを初期化するだけ。データ文や代入文で後から値を上書きできる。

default の例

```
param mininv {p in PROD, t in 0..T}
   default frac * market[p,t+1];

# データ文で例外的な値を
# 上書きできる
param mininv :=
   bands 2 3000
   coils 2 2000 ;
```

再帰的定義（移動平均）

```
param mininv {p in PROD, t in 0..T} =
  if t = 0 then inv0[p]
  else 0.5 * (mininv[p,t-1]
       + frac * market[p,t+1]);

# 循環参照はAMPLがエラーで検出
```

■ = 句や default 句の後には、制限句と同様の追加チェック（例：, >= 0）も付けられる。

乱数関数で試験用データを作る

- 計算されるパラメータの一種。式に組込みの乱数生成関数（Normal・Uniform 等）を使う。
- モデル開発の初期段階や、異なる定式化・ソルバーの比較実験に便利。
- option randseed n; で種を変更できる。0 を指定すると毎回異なる系列になる。
- reset data コマンドで新しい乱数の標本を取り直せる。

正規分布に従う利用可能時間

```
param avail_mean > 0;
param avail_variance
   > 0, < avail_mean/2;
param avail =
  max(Normal(avail_mean,
             avail_variance), 0);
```

一様分布に従う需要シェア／randseed

```
param share {DEST} = Uniform(0,100);
param tot_sh =
   sum {j in DEST} share[j];

option randseed n;
```

0/1

7-7 論理パラメータ ／ 7-8 記号パラメータ

binary と symbolic

```
param promote {PROD,1..T+1} binary;
```

- AMPL に専用の論理型はないが、binary パラメータ（0/1）で代用できる。
- 算術式が論理式の場所に現れると、0 以外は真、0 は偽と解釈される（暗黙変換に注意）。

```
param entr symbolic in INTER;
```

- symbolic を付けると文字列値を表せる（数値でも文字列でもよいが、文字列は算術に参加できない）。
- 用途：特別な集合要素の指定、集合要素への説明文字列の付与など。

promote を条件式で使う

```
param mininv {p in PROD, t in 0..T} =
  (if promote[p,t] then fr_pro
   else fr_reg) * market[p,t+1];
```

発地の集合に説明文字列を添える

```
set ORIG;
param orig_name {ORIG} symbolic;
param supply {ORIG} >= 0;
```

木構造を集合とパラメータで表す

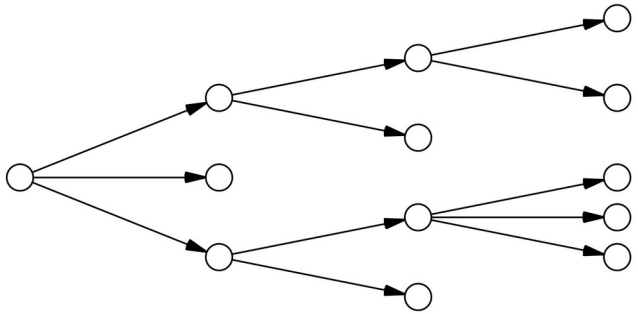


図7-1　根から伸びる木

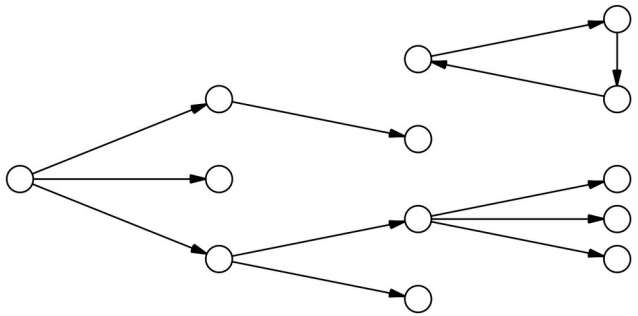


図7-2　データ誤りで生じた切り離された閉路

木構造の宣言

```
set NODES;
param Root symbolic in NODES;
param pred {i in NODES diff {Root}}
   symbolic in NODES diff {i};
```

- Root 以外のすべての節点 i は、先行節点 pred[i] を一つもつ。
- 節点の「深さ」＝根に到達するまでに遡る先行節点の数（根の深さは 0）。
- depth[i] は pred を使った再帰的な param 定義で計算できる。
- 閉路を含む誤ったデータでは、depth の再帰的定義が循環参照となり計算できない。

まとめ —第 7 章　パラメータと式

1	**param 宣言**	スカラーから多次元・疎な集まりまで、添字式で自在に定義できる。
2	**演算子と反復式**	算術・論理演算子に加え、sum / prod / exists / forall で代数記法の Σ・Π を一般化。
3	**if-then-else**	論理式の真偽で値を選ぶ条件式。制約や計算されるパラメータの記述を簡潔にする。
4	**制限句と check 文**	宣言時の制限句とデータ全体を検査する check 文で、誤りを早期に発見する。
5	**=／default／binary／symbolic**	計算・初期化・真偽・文字列など、パラメータの多様な役割を使い分ける。

次章：第 8 章「変数」へ続く

AMPL：数理計画のためのモデリング言語



第8章

線形計画問題：変数・目的・制約

var 宣言・線形式・目的関数・制約の書き方を体系的に学ぶ

AGENDA

本章の内容

8.1	変数	var 宣言、上限・下限、定義された変数、整数・バイナリ変数
8.2	線形式	AMPL が線形と認識する式の形と、等価な書き換え規則
8.3	目的	minimize / maximize の宣言と、複数目的の切り替え
8.4	制約	subject to の宣言、添字づけ、二重不等式、線形化

INTRODUCTION

線形計画問題とは

変数は連続な範囲から値をとり、目的と制約は変数の線形関数のみを使う――これが線形計画問題（LP）の要件である。

- ここまでの全ての例（生産計画・配合・輸送・部屋割当）はすべて LP
- 線形関数の単純さが定式化・解釈・分析を容易にする
- 制約と変数が数千個に及んでも、最適解を正確かつ迅速に発見できる
- ネットワークなど特別な構造をもつ LP には専用の記法もある（第15〜17章）

本章で学ぶ4本柱

1. var（変数）
2. 線形式の規則
3. minimize / maximize
4. subject to（制約）

8.1 変数

var 宣言の基本構文

- **var 宣言の構文は param 宣言と同一**
  - 違いはキーワードのみ： param → var
- **>= と <= の句は、変数の下限・上限（一種の制約）を課す**
  - 以前に定義した集合・パラメータと、現在のダミー添字による任意の算術式を指定できる
- **< 、 > 、 <> は変数宣言に使えない**
  - 線形計画では「< 3」に意味がない（2.99999…に望むだけ近づけてしまう）

```
# パラメータ宣言（第7章）
param market {PROD};

# 変数宣言：param → var のみ変更
var Make {p in PROD}
   >= 0, <= market[p];
```

- Make[p] は集合 PROD の各要素 p ごとに1つ作られる添字づけられた変数の集まり
- >= 0：すべての Make[p] は非負
- <= market[p]：各 p について Make[p] は market[p] を超えられない

上限・下限の指定パターン

直接指定

```
var Make {p in PROD}
   >= 0, <= market[p];
```

非負性を >= 0 で直接指定する、最も一般的な形。

間接指定

```
param f_min {FOOD} >= 0;
param f_max {j in FOOD} >= f_min[j];

var Buy {j in FOOD}
   >= f_min[j], <= f_max[j];
```

図5-1 配合モデル：下限・上限をパラメータで間接的に指定する形。

>= と <= に続く値が、変数の下限（lower bound）と上限（upper bound）である。

var 宣言に上下限を置けばモデルは短く明快になるが、最適解自体は変わらない。ソルバーの表現によらず、AMPL はすべての上下限を自動的に識別する。

定義された変数（= 句）

- var 宣言の = 句はパラメータと同様に「定義」を生む
- 右辺には集合・パラメータに加えて既存の変数も使える
- = の右の線形式が、目的・制約中のその変数の出現箇所すべてに代入される
- 新しい制約が追加されるわけではない
- 目的の構造が明快になり、display で内訳を手軽に確認できる

```
var Total_Revenue =
   sum {p in PROD, t in 1..T}
      sum {a in AREA[p]}
         revenue[p,a,t]*Sell[p,a,t];

maximize Total_Profit:
   Total_Revenue - Total_Prod_Cost
      - Total_Inv_Cost;
```

```
ampl: display Total_Revenue, Total_Prod_Cost,
ampl?    Total_Inv_Cost;
Total_Revenue = 801385
Total_Prod_Cost = 285643
Total_Inv_Cost = 1221
```

その他の宣言オプション

1

:= と default（初期値）

ソルバー起動時の開始解を与える。LP のソルバーは開始解をうまく活かせないことが多く、非線形計画（第18章）でより有用。

2

integer（整数変数）

どの最適解でも整数値をとらねばならない。含むモデルは整数計画問題（第20章）となる。

3

binary（バイナリ変数）

0 と 1 の値しかとれない。整数計画問題の代表的な部品となる。

一時的に変数を固定したいだけなら宣言はそのままにして fix コマンド（11.4節）を使う。

恒久的にデータの値へ固定するなら、= の右辺に変数を含めず param 宣言を使うべきである。

線形式とは何か

変数が1単位増減するたびに式の値が一定量だけ増減する ―― これが「その変数について線形」の定義。すべての変数について線形な式を線形式と呼ぶ。

AMPL が線形と認識する項

```
constant-expr
variable-ref
(constant-expr) * variable-ref
```

この形の項の任意の和が線形式である。constant-expr は変数を含まない算術式、variable-ref は（添字付き可）変数への参照。

線形式の実例（図6-3）

```
avail[t]

Make[p,t] + Inv[p,t-1]

sum {p in PROD}
   (1/rate[p]) * Make[p,t]

sum {a in AREA[p]} Sell[p,a,t]
   + Inv[p,t]
```

線形式の書き換え規則

除算 ＝ 逆数の乗算

```
Make[p,t] / rate[p]
```



```
(1/rate[p]) * Make[p,t]
```

乗算の順序は自由

```
revenue[p,a,t] * Sell[p,a,t]
```



```
Sell[p,a,t] * revenue[p,a,t]
```

乗除算は sum に分配できる

```
mt_t * sum{...}( ... )
```



```
sum{...}( ... ) / t_mt
```

AMPL はこれらの等価な式を自動的に「線形の項の単純な和」へ変換する。演算子の優先順位（表A-1）で結果が同じになるなら括弧は省略できる。

if-then-else と非線形の扱い

線形になる場合

then と else の式がともに線形で、if と else の間の論理式に変数が現れなければ結果は線形。

```
Make[j,t] + (if t = first(WEEKS)
   then inv0[j]
   else Inv[j,prev(t)])
```

線形にならない場合

線形式中の変数は、他のいかなる演算子の被演算子にも、いかなる関数の引数にもなれない。

- 反復演算子： max, min, abs, forall, exists
- 数値関数： ^（べき乗）, sqrt, log, cos など

線形式の一般形： constant-expr ／ var-ref ／ (constant-expr)*(linear-expr) ／ (linear-expr)/(constant-expr) ／ if logical-expr then linear-expr else linear-expr の和。この処理は AMPL が自動で行う。



8.3 目的

minimize / maximize の宣言

- キーワード minimize または maximize、名前、コロン、線形式で構成
- 目的の名前は代数モデル中でそれ以上の役割をもたない（列単位の宣言を除く）
- コマンド環境では目的の名前がその値を指す
- 1つの LP は目的関数を1つもたねばならないが、モデルには複数の objective 宣言があってよい
- 添字式を付ければ、添字づけられた目的の集まりも定義できる

```
minimize Total_Cost:
   sum {j in FOOD} cost[j]*Buy[j];
```

```
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]);
```

8.3 目的

objective コマンドで目的を切り替える

図2-1 配合モデル：総費用の最小化とは別に、総ナトリウム最小化の目的を追加して比較する。

```
minimize Total_NA:
   sum {j in FOOD} amt["NA",j]*Buy[j];
```

```
ampl: solve;   # 既定：先頭の目的
Objective = Total_Cost

ampl: objective Total_NA;
ampl: solve;
ampl: display Total_Cost;
```

目的を切り替えた結果

Total_Cost 最小化

```
Total_Cost = 118.06    Total_NA = 50000
```

Total_NA 最小化

```
Total_NA = 48186    Total_Cost = 123.63
```

ナトリウムを約1800減らすと費用は約 $5.50 押し上げられる

subject to の宣言

- キーワード subject to、名前、コロンで始まる（subject to 自体は省略可）
- コロンの後には制約の代数的記述（線形式 と 線形式 を等号・不等号で結んだもの）が続く
- 制約名の後に添字式を与えると、添字づけられた制約の集まりになる
- 制約名はコマンド環境で双対値の指定などに使われる（第14章）

```
subject to Time:
   sum {p in PROD} (1/rate[p])
      * Make[p] <= avail;
```

```
subject to Time {t in 1..T}:
   sum {p in PROD} (1/rate[p])
     * Make[p,t] <= avail[t];
```

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

制約の応用パターン

条件付き添字

```
subject to Time {t in 1..T:
                 avail[t] > 0}: ...
```

二重不等式

```
subject to Diet {i in NUTR}:
  n_min[i] <= sum{j in FOOD}
   amt[i,j]*Buy[j] <= n_max[i];
```

擬似添字 *{if avail > 0}* は単一の制約を条件付きで生成／非生成にする。

非線形 → 線形への書き換え

```
# 線形でない（変数の和で除算）
Make[p] / sum{q in PROD} Make[q]
   <= max_frac;

# 書き換えれば線形になる
Make[p] <= max_frac
   * sum {q in PROD} Make[q];
```

前処理（presolve）

ソルバーに渡す前に AMPL が制約を自動簡約：固定変数の除去、単一変数制約の上下限化、冗長な制約の除去など。

SUMMARY

第8章のまとめ

1

var 宣言は param と同じ構文

>= と <= が変数の下限・上限を課す。= 句は変数を「定義」し、目的・制約中に代入される。

2

線形式は決まった形の項の和

constant-expr、variable-ref、その積の和のみ。max/min/abs 等の引数に変数を置くと非線形になる。

3

目的は minimize / maximize で宣言

1つの LP に目的は1つ必須だが宣言は複数可。objective コマンドで最適化対象を切り替えられる。

4

制約は subject to、添字づけ自在

二重不等式や条件付き添字も使える。除算や比の形は乗算に書き換えて線形化する。

次章：第9章へ続く

AMPL：数理計画のためのモデリング言語

第 9 章

データの指定

集合とパラメータの値をどう表すか ― data コマンドとデータ文の書式

モデルとデータの分離 ｜ リスト形式 ｜ 表形式 ｜ read コマンド

本章の内容

Agenda

9.1 整形されたデータ：data コマンド

モデルモードとデータモードの切替、構文ルール、複数ファイル

9.2 リスト形式のデータ

一次元〜多次元の集合・パラメータ、テンプレートによるスライス

9.3 表形式のデータ

二次元の表、転置・分割、高次元データのスライス表・多次元表

9.4 データ文のその他の機能

既定値（default）、添字付き集合の集まり、変数の初期値

9.5 整形されていないデータの読み込み：read コマンド

item-list による柔軟な値の取り込み

モデルモードとデータモードの切替

9.1

- param・var の宣言や solve・display コマンドは「モデルモード」で実行される（作業の大半で標準）
- モデルモードは集合・パラメータの長い値のリストを読むには不便
- data キーワードで開始すると「データモード」に入り、データ文を読む
- data diet.dat; のようにファイル名を付ければそのファイルから読む
- ファイル名なしの data; は以降の入力をデータ文とみなす（対話的に使える）
- set/param で始まらない文（display など）や model コマンドでモデルモードに復帰

対話セッション例

```
ampl: model dietu.mod;
ampl: data;
ampl data: set MINREQ := A B1 B2 C CAL;
ampl data: set MAXREQ := A NA CAL;
ampl data: display NUTR;
set NUTR := A B1 B2 C CAL NA;
ampl:

ampl: data diet.dat;
  # ファイル名を指定して読む場合
```

データモードの構文ルール

9.1

- 空白・タブ・改行はまとめて 1 個のスペース、区切りのコンマも無視される → 読みやすい表に整形できる
- 文字列は通常、引用符で囲む必要はない
- 英字・数字・下線・ピリオド・+・- 以外の文字を含む文字列は引用符が必須（例：'A&P'）
- 引用符自体を含む文字列は種類を変えるか二重にする
- 数値に見える文字列（"+1"、"3e4"）も数値と区別するため引用符が必要

引用符の例

- "DOMINICK'S"（二重引用符で単一引用符を含める）
- 'DOMINICK''S'（引用符を内側で二重化）

同じ内部表現の数値は同じ集合要素

```
2  =  2.00  =  2.e0  =  0.02E+2
```

いずれも同一の値として扱われる

複数データファイルと重複の検査

9.1

- 複数の data コマンドを使えば、モデルの構成要素に任意の数のデータファイルから値を割り当てられる
- AMPL はどの構成要素にも値が 2 回以上割り当てられていないかを常に検査する
- 重複する割当はエラーとして指摘される（下の例）
- シナリオを変えて一部データだけ更新したい場合は、reset data や update data コマンドを先に与えれば新しい値を受け入れさせられる（第11.3節）
- let コマンドで集合・パラメータの値を直接割り当てる方法もある（第11.3節）

重複エラーの例

```
duplicate member coils
  for set PROD
context: set PROD :=
  bands coils plate coils
  >>> ; <<<
```

ポイント

ファイル数によらず「重複割当なし」が保証される。意図的な再割当には reset data / update data / let を使う。

一次元の集合とパラメータのリスト

9.2

モデル + データ

```
set PROD;
param rate {PROD} > 0;

set PROD := bands coils plate ;

param rate :=
  bands   200
  coils   140
  plate   160 ;

# 余分な空白や改行は無視される
```

- 集合の指定は要素の列挙のみ。パラメータは各要素の後に値を続けるだけ
- 空白・改行は無視されるため、表のように整形して読みやすくできる
- 集合の要素はすべて異なる必要があり、重複は AMPL が警告する
- 同様にパラメータへの二重の値割当てもエラーになる
- 要素の空リスト（:= の直後にセミコロン）で集合を空と指定できる
- ordered / circular 属性の集合は、要素の順序がデータ文の順で決まる

二次元集合：組のリストとテンプレート

9.2

組のリスト（LINKS）

```
set ORIG;  set DEST;
set LINKS within {ORIG,DEST};

set LINKS :=
  (GARY,DET) (GARY,LAN)
  (CLEV,FRA) (PITT,FRA) ... ;
```

テンプレートでスライス

```
set LINKS :=
  (GARY,*) DET LAN STL LAF
  (CLEV,*) FRA DET LAN WIN ...
  (PITT,*) FRA WIN STL FRE ;
```

データテンプレートとスライス

- 対に似ているが一方の成分が * に置き換わった式を「データテンプレート」と呼ぶ
- (GARY,*) の後にリストが続き、各項目が * に代入されて対が生成される
- こうしてできた対の集まりが、その次元における集合の「スライス」をなす
- 第 1 成分固定なら (GARY,*)、第 2 成分固定なら (*,FRA) のように使う
- 対の中の要素順には意味がある：第 1 要素は ORIG、第 2 要素は DEST から
- 対そのものは任意の順序で書いてよい（* を含まない対は自分自身のみを指定）

二次元パラメータのリスト

9.2

- 集合 LINKS で添字づけられたパラメータ cost も、集合と同じ形式にパラメータ値を加えて指定する
- スライスのリストも同様に拡張され、暗黙の各集合要素の後にパラメータ値を置く
- パラメータのテンプレートは角括弧 [ ] で囲む（集合の丸括弧 ( ) と区別するため）
- [GARY,*] の後の DET 14 は、cost["GARY","DET"] に 14 を与えることを意味する
- 対のリストは、既定テンプレート [*,*] を使った特別な場合とみなせる

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

param cost :=
  [GARY,*] DET 14  LAN 11
           STL 16  LAF 8
  [CLEV,*] FRA 27  DET  9
           LAN 12  WIN 9
           STL 26  LAF 17
  [PITT,*] FRA 24  WIN 13
           STL 28  FRE 99 ;
```

三次元以上への拡張と一般規則

9.2

1次元スライス（ROUTES）

```
set ROUTES within {ORIG,DEST,PROD};
set ROUTES :=
  (CLEV,*,bands) FRA DET LAN ...
  (GARY,*,coils) LAN STL LAF ;
```

2次元スライス（* が2つ）

```
set ROUTES :=
  (*,FRA,*) CLEV bands
            CLEV coils
            PITT bands ;
# * の数だけ後続の要素が対で続く
```

一般規則（n 次元テンプレート）

- n 次元の集合・パラメータのテンプレートは n 個の項目（正しい集合要素または *）を持つ
- 集合のテンプレートは組と同様に丸括弧、パラメータのテンプレートは角括弧で囲む
- テンプレートの後には、* 一つにつき一つの集合要素（パラメータなら値も）が続く
- * の数が異なるテンプレートを同じデータ文で混在させてもよい
- テンプレートがない箇所は、すべて * のテンプレートが仮定される

集合とパラメータを結合したリスト

9.2

冗長な書き方

```
set PROD := bands coils plate ;
param rate :=
  bands 200 coils 140 plate 160 ;
```

集合名を param 文に含めて重複を回避

```
param: PROD: rate :=
  bands 200  coils 140  plate 160 ;
```

複数パラメータをまとめる

```
param rate   := bands 200 coils 140 plate 160 ;
param profit := bands  25 coils  30 plate  29 ;
param market := bands 6000 coils 4000 plate 3500 ;

# ↓ 一つの文にまとめられる
param: rate profit market :=
  bands 200 25 6000
  coils 140 30 4000
  plate 160 29 3500 ;
```

集合＋複数パラメータも一括可能

```
param: PROD: rate profit market := ... ;
```

欠損値のドット（.）表記

9.2

- n_min は MINREQ、n_max は MAXREQ という重なりうる異なる集合で添字づけられる
- MINREQ にない NA については n_min[NA] は定義されない → ドット（.）を明示的に置く
- 空欄のままにはできない：空白は無視されるため次の値と誤って結び付く
- ドットの代わりに 0 を置くのも誤り。未定義のまま参照するとエラーになる

アクセス時のエラー例

```
error processing param n_min:
  invalid subscript n_min['NA'] discarded.
```

栄養素	n_min	n_max
A	700	20000
C	700	.
B1	0	.
B2	0	.
NA	.	50000
CAL	16000	24000

表形式データの基本

9.3

- 行のラベルが第1添字、列のラベルが第2添字を与える
- パラメータ名の後にコロン、列ラベルの後に := を置くと AMPL が表と認識する
- cost["GARY","FRA"] は 39 に設定される

cost	FRA	DET	LAN	WIN	STL	FRE	LAF
GARY	39	14	11	14	16	82	8
CLEV	27	9	12	9	26	95	17
PITT	24	14	17	13	28	99	20

図 3-1b：param cost: FRA DET LAN WIN STL FRE LAF := ... ;

転置：(tr)

添字集合の一方だけが大きい場合、小さい方の集合を列に置くため表を転置できる。

```
param cost (tr):
  GARY CLEV PITT :=
  FRA   39   27   24
  DET   14    9   14
  ... ;
```

(tr) は列ラベルが第1添字、行ラベルが第2添字を表す

大きな表の分割と集合の +/- 表

9.3

- 改行は無視されるため、各行を複数行に折り返して幅を抑えられる
- 表を列方向にいくつかの部分表へ分割してもよい。新しい部分表の先頭はコロンで示す
- LINKS のような疎な添字集合では、表中にドット（.）を置いて「値なし」を示す
- 集合自体も同じ形の表で指定でき、要素なら +、非要素なら - を置く

列方向の分割（例）

```
param cost: FRA DET LAN WIN :=
   GARY  39  14  11  14
   CLEV  27   9  12   9
        : STL FRE LAF :=
   GARY  16  82   8
   CLEV  26  95  17 ;
```

LINKS	FRA	DET	LAN
GARY	-	+	+
CLEV	+	+	+
PITT	+	-	-

+ は集合の要素である対、- は要素でない対を示す（パラメータの表形式はそのまま集合にも使える）

高次元データの表：スライスと多次元表

9.3

2次元スライスの並び

```
param cost :=
 [*,*,bands]: FRA DET LAN WIN STL FRE LAF :=
  CLEV  27  9  12  .  26  .  17
  PITT  24  .  .  13  28  99  .
 [*,*,coils]: FRA DET LAN WIN STL FRE LAF :=
  GARY   .  .  11  .  16  .  8
  ... ;
```

テンプレートで配置を精密指定

```
[*,:,*] : FRA DET LAN ... :=
 CLEV bands  27   9  12 ...
# * は左（行）、: は上（列）に来る添字
```

多次元表の考え方

- 2次元を超えるデータは、二次元の「スライス」表の並びとして指定できる（テンプレートに * が2つ）
- 左右の見出しに2つ以上の添字を置き、単一の表で多次元データを表すこともできる
- 見出し行それぞれの前にコロンを置き、:= は最後の見出し行の後にのみ置く
- 既定では左→上の順に添字が取られる。(tr) を付けると上→左の順に転置される
- 表現の選び方はデータの意味に影響しない → 読みやすさ・簡潔さで自由に選べる

既定値（default 句）

- 値の与え忘れ・与えすぎはどちらもエラーになる（solve 時などに検出）
- 同じ値が何度も現れるなら default 句で繰り返し指定を避けられる
- 添字づけられた集まり全体に一様な値を与える場合にも便利（例：供給量をすべて 1 に）
- default はモデルの param 宣言中にも書ける。既定値を他のデータに依存させたい場合はこちらを使う

データ文での default

```
param cost default 9999 :=
 [*,*,bands]: FRA DET LAN ... :=
  CLEV 27 9 12 . 26 . 17
  PITT 24 . . 13 28 99 .
 [*,*,coils]: ... ;
```

24

個の 9999 が自動的に割り当てられる

```
param supply default 1;
param demand default 1;
```

割当問題：供給・需要を一律 1 に

添字付き集合の集まり／変数の初期値

9.4

集合の添字づけられた集まり

```
set PROD;
set AREA {PROD};
set AREA[bands] := east north ;
set AREA[coils] := east west export ;
```

- 集合名にキーワード set の後で添字を付ける以外、通常の集合データ文と同じ形式が使える
- 空にしたい要素は空リストで指定。データを与えない要素は、モデル側で default {} を宣言すれば警告を避けられる

変数の初期値

```
var Trans:  FRA  DET  LAN  WIN  STL  FRE  LAF :=
  GARY  100  100  800  100  100  500  200
  CLEV  900  100  100  500  500  200  200
  PITT  100  900  100  500  100  900  200 ;
```

- param データ文と同じ形式で、パラメータ名の代わりに変数名・制約名を使えばよい（先頭を var にしてもよい）
- 初期値はソルバーへの出発点になる。非線形最適化で特に有用（第18章）

read コマンド：整形されていないデータ

9.5

- ファイルが集合名・パラメータ名・:= を含まず、値だけが順序正しく並んでいる場合に便利
- 基本形：read item-list < filename ;　値は現れる順にリストの項目へ割り当てられる
- item は「パラメータ」「{添字式} パラメータ」「{添字式} (item-list)」のいずれか（入れ子も可）
- ファイル名を省くと現在の入力（対話プロンプトなど）から読む。"-" は標準入力を表す

基本例：週次データ

```
param T > 0;
param avail {1..T} >= 0;

read T, {t in 1..T} avail[t]
  <week_data.txt;

# week_data.txt:  4  40 40 32 40
```

入れ子の item-list

```
read {p in PROD}
  (prodcost[p], invcost[p],
   {t in 1..T} (revenue[p,t],
                market[p,t]))
  <cost_data;
```

順序に関する注意

文字列で添字づけられた集合は、要素の内部順序で読まれる。順序を確実にするには ordered / ordered by を使うか、ファイル中に添字自体を明示する。

✓ まとめ

1. **data コマンドでモデルモードとデータモードを切替**
   宣言（モデル）と値（データ）を明確に分離して記述できる

2. **リスト形式はテンプレート（*）でスライスを簡潔に表現**
   1次元〜多次元の集合・パラメータに一貫した構文を適用

3. **表形式は2次元以上の添字を読みやすく配置**
   転置 (tr)、部分表への分割、+/- 記法、多次元見出しが使える

4. **default・添字付き集合の集まり・変数初期値など補助機能が豊富**
   繰り返しの排除や、ソルバーへの出発点指定に活用できる

5. **read コマンドで整形されていない外部データも柔軟に取り込める**
   item-list の入れ子で、ファイルの並びに合わせた読み込みが可能

AMPL：数理計画のためのモデリング言語



第10章

データベースアクセス

table 宣言でAMPL の集合・パラメータ・変数を関係データベースの表に対応づける

table 宣言

read table / write table

ODBC・Access・Excel・テキスト

本章の内容

AGENDA

1. **データ対応の一般原則**
   AMPLの実体と関係表の列の対応
2. **テーブル操作文の例**
   table宣言・read table・write table
3. **関係表からのデータの読み込み**
   キー列・データ列・集合の読込
4. **関係表へのデータの書き出し**
   data-spec / key-spec からの推論
5. **同じ表の読み書き**
   IN・OUT・INOUTの使い分け
6. **表と列の添字づけられた集まり**
   テーブル配列と2次元表の入出力
7. **標準および組込みのテーブルハンドラ**
   ODBC・テキスト・バイナリ・SQL

AMPLの実体と関係表の対応

10.1

diet.mod

```
set FOOD;

param cost {FOOD} > 0;
param f_min {FOOD} >= 0;
param f_max {j in FOOD}
   >= f_min[j];
```

- 集合 FOOD の要素 → 表の「キー列」
- パラメータ cost / f_min / f_max → 各1本の「データ列」
- 1行 = 集合の1要素（1食品）のすべてのデータ

FOOD	cost	f_min	f_max
BEEF	3.19	2	10
CHK	2.59	2	10
FISH	2.29	2	10
HAM	2.89	2	10
…	…	…	…

キー列 FOOD の項目は一意 = 各キー値がちょうど1行を特定する

行 = レコード　列 = フィールド

データベース用語では、表の各行はレコード、レコード内の各項目はフィールドと呼ばれる。

レコード単位のデータ入力

10.1

- キー列の項目は集合の要素と同様に一意でなければならない
- データベースソフトでは、表の行は「レコード」、列は「フィールド」とみなされる
- データ入力フォームには列ごとに1つの入力欄が用意される
- 配合問題のフォーム（Microsoft Access）では、表の各行に対応するレコードを、フォーム下部の操作部で1件ずつ入力・閲覧できる

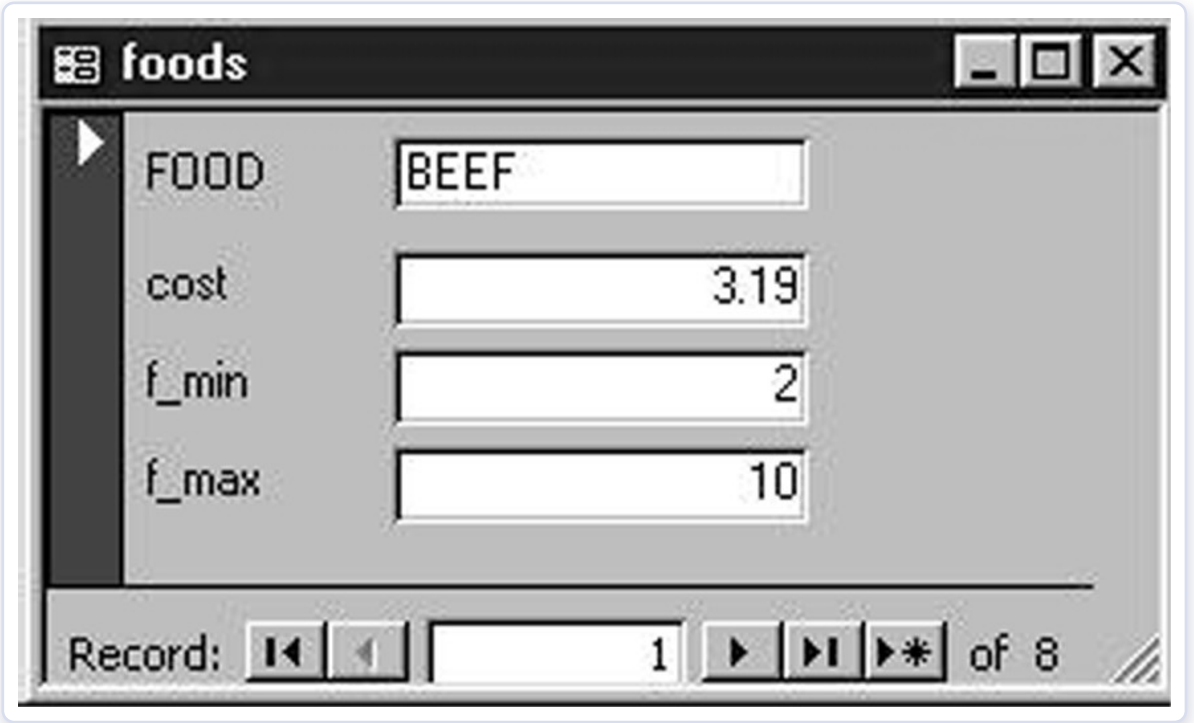


図10-1　Accessのデータ入力フォーム

多次元の実体 → 複数のキー列

10.1

steelT.mod

```
set PROD;       # products
param T > 0;    # number of weeks

param market {PROD,1..T} >= 0;
param revenue {PROD,1..T} >= 0;
var Make {PROD,1..T} >= 0;
```

- PROD と 1..T の直積で添字づけられた実体 → キー列が2つ (PROD, TIME)
- 各次元につき1つのキー列を割り当てる
- 見出し TIME は式 1..T を表すために考案した名前

PROD	TIME	market	revenue	Make	Sell
bands	1	6000	25	5990	6000
bands	2	6000	26	6000	6000
coils	1	4000	30	1407	307
coils	2	2500	35	1400	2500

2つのキー列の順序対は一意

集合 {PROD,1..T} で対が一意であるのと同様、表でも ("bands",1) や ("coils",3) のような組はちょうど1行に対応する。

データで決まる集合との対応

10.1

transp3.mod

```
set LINKS within {ORIG,DEST};
param cost {LINKS} >= 0;
var Trans {LINKS} >= 0;
```

- LINKS は1次元集合から構成されるのではなく、データとして読まれる対の集合
- 対応する関係表もキー列を2つもつ（ORIG, DEST）
- 表には ORIG・DEST の全組合せではなく、実際に集合 LINKS に属する組だけが行として現れる

ORIG	DEST	cost	Trans
GARY	DET	14	0
GARY	LAF	8	600
CLEV	DET	9	1200
PITT	STL	28	900
PITT	WIN	13	0

網羅的な直積表ではなく「実在する組だけ」を並べる疎な表

table宣言の構文構造

10.2

```
table table-name IN string-list :  key-spec, data-spec, data-spec, ... ;
```

テーブル名

AMPL内でこの表を指す名前（例：Foods）

IN / OUT

既定の読み書き状態。省略時は INOUT

string-list

外部ファイル・表の場所（ハンドラ固有）

key-spec

キー列 [...] と集合との対応（<-, ->, <->）

data-spec

データ列とパラメータ・変数・式との対応

コロンより前＝表の所在（テーブル名・*IN/OUT・外部ファイル*）　／　*コロンより後＝列とAMPLの実体との対応*

同じ対応、異なる外部形式

10.2

Access

```
table Foods IN "ODBC" "diet.mdb":
   FOOD <- [FOOD], cost, f_min,
f_max;
```

diet.mdb

Excel

```
table Foods IN "ODBC" "diet.xls":
   FOOD <- [FOOD], cost, f_min,
f_max;
```

diet.xls

テキスト

```
table Foods IN:
   FOOD <- [FOOD], cost, f_min,
f_max;
```

Foods.tab

- コロン前の情報だけが変わる：ODBCハンドラ経由か、既定のテキストハンドラかを指定
- 3つとも AMPL 側のテーブル名は同じ Foods → read table Foods は共通
- 詳細を省略すると既定でテキストファイル Foods.tab が使われる

read table でAMPLに読み込む

amplセッション

```
ampl: model diet.mod;
ampl: table Foods IN "ODBC" "diet.mdb":
ampl?    FOOD <- [FOOD], cost, f_min, f_max;
ampl: read table Foods;
ampl: display cost, f_min, f_max;
:      cost f_min f_max    :=
BEEF   3.19    2    10
CHK    2.59    2    10
FISH   2.29    2    10
HAM    2.89    2    10
  ...
;
```

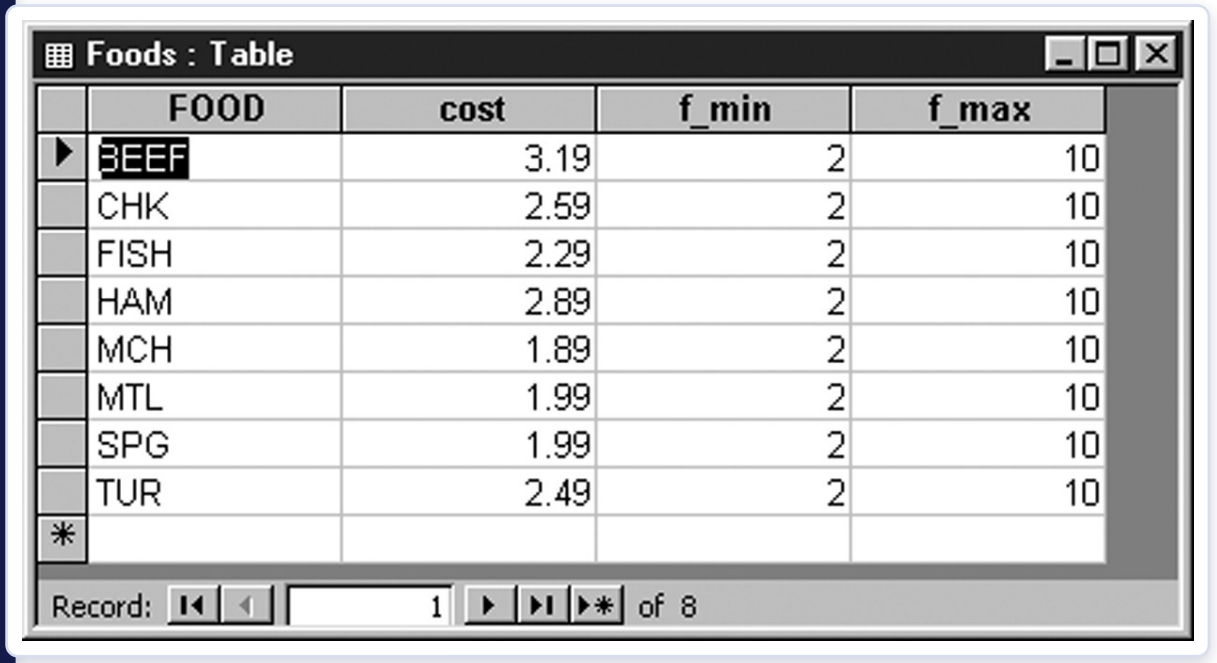


図10-2　Accessの関係表

入力と出力を1つの宣言に


```
table Foods "ODBC" "diet.mdb":
   [FOOD] IN, cost IN,
   f_min IN, f_max IN, Buy OUT,
   Buy.rc ~ BuyRC OUT,
   {j in FOOD} Buy[j]/f_max[j]
      ~ BuyFrac OUT;
```

Foods : Table

FOOD	cost	f_min	f_max	Buy	BuyRC	BuyFrac
BEEF	3.19	2	10	5.360613811	3.05311E-16	0.536061381
CHK	2.59	2	10	2	1.18884058	0.2
FISH	2.29	2	10	2	1.144407502	0.2
HAM	2.89	2	10	10	-0.30265132	1
MCH	1.89	2	10	10	-0.5511509	1
MTL	1.99	2	10	10	-1.32890026	1
SPG	1.99	2	10	9.306052856	-2.0123E-15	0.930605286
TUR	2.49	2	10	2	2.731619778	0.2

Record: 1 of 8

図10-4　出力列が加わったAccessの関係表

- IN / OUT を表全体ではなく列ごとに指定できる
- ~ 演算子で AMPL の式（Buy.rc など）をデータベースの正しい列名に結びつける
- read table → IN列を読み、write table → OUT列に書き出す

データ読み込みの2つの基本形

10.3

パラメータのみ読む

```
table Foods IN:
   [FOOD], cost, f_min, f_max;
```

- 集合 FOOD はモデルまたは他の表であらかじめ定義されている
- キー列の項目を各パラメータの添字として値を割り当てるだけ

集合とパラメータを同時に読む

```
table Foods IN:
   FOOD <- [FOOD], cost, f_min, f_max;
```

- 矢印 <- はキー列 → 集合への読み込み方向を示す
- key-spec を短縮形 [FOOD] IN と書くこともできる（集合名=キー列名の場合）

列名の対応づけとダミー添字による調整

10.3

列名が違う

*AMPL*側と表側で名前が食い違う場合

```
f_min ~ lowerlim, f_max ~ upperlim
```

番号の基準が違う

表は0始まり、モデルは1始まりなどのずれを補正

```
[p ~ PROD, t ~ TIME], market[p,t+1] ~ market
```

添字の順序が違う

同じ集合でも添字の並びが異なる場合

```
market, revenue[t,p] ~ revenue
```

1つの値が複数列に分割

系列ごとに別列で保持されている場合

```
revenue["bands",t] ~ revbands, revenue["coils",t] ~ revcoils
```

データ指定(data-spec)から行を推論

10.4

```
table Foods OUT: [FoodName],
   f_min, Buy, f_max;
```

- key-spec が単なる [キー列名] のリストのとき、display コマンドと同様に動く
- data-spec が示す（あるいは含意する）添字集合の和で、書き出す行が決まる
- 挙げる項目はすべて同じ次元でなければならない

FoodName	f_min	Buy	f_max
BEEF	2	5.36061	10
CHK	2	2	10
HAM	2	10	10
SPG	2	9.30605	10

write table Foods; → FOOD の各要素につき1行が生成される

キー指定(key-spec)から行を推論


```
table Foods OUT:
   FOOD -> [FOOD],
   f_min, Buy, f_max;
```

- 矢印 -> は AMPL の集合 → キー列へ書き出す方向
- 明示した集合の各要素につき1行を書き出す（data-specの添字集合からの推論ではない）
- 同じ集合名がキー列名なら [FOOD] OUT と略記できる

エラーになる例

```
table Nutrs OUT: [Nutrient],
   Diet.lslack, Diet.ldual, ...;
# Column "Diet.lslack" contains '.'
```

~ で列名を与えて解決

```
table Nutrs OUT: [Nutrient],
   Diet.lslack ~ lb_slack,
   Diet.ldual ~ lb_dual,
   Diet.uslack ~ ub_slack,
   Diet.udual ~ ub_dual;
```

同じ表を読み書きする

10.5

- 方法A：読み込み用と書き出し用、2つの table 宣言を使う
- 方法B：1つの table 宣言に読み書きの状態をまとめる

指定	read table	write table
<- / IN	読む	—
-> / OUT	—	書く
<-> / INOUT	読む	書く

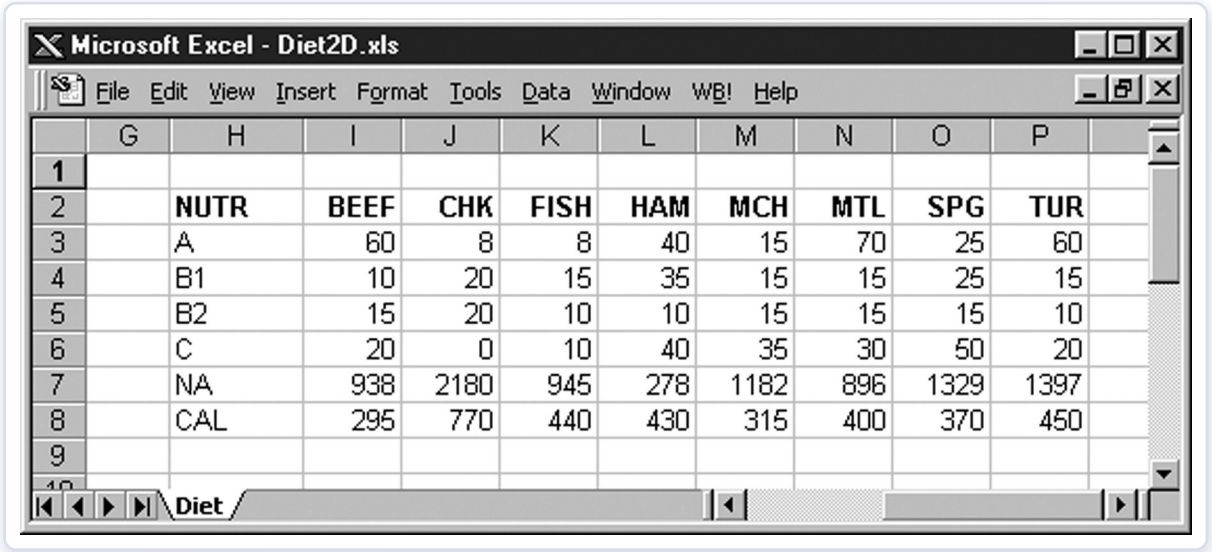
単一の宣言

```
table Foods "ODBC" "diet1.mdb" "Diet":
   FOOD <- [FoodName],
   cost IN, f_min IN,
   Buy OUT, f_max IN;
```

- read table Foods → IN・INOUT状態の列だけ読む（FoodName, cost, f_min, f_max）
- write table Foods → OUT・INOUT状態の列だけ書く（Buy）
- 既存の表・ファイル全体を上書きするのは、全列がOUTのときだけ

テーブル・列の添字づけられた集まり

- table 宣言も {indexing-expr} で添字づけできる（例：食品ごとに別々の表へ出力）
- data-spec も {j in FOOD} <…> の形で列の集まりを生成できる
- 二次元の表計算データ（行×列）を読み書きするのに特に便利



	G	H	I	J	K	L	M	N	O	P
1										
2		NUTR	BEEF	CHK	FISH	HAM	MCH	MTL	SPG	TUR
3		A	60	8	8	40	15	70	25	60
4		B1	10	20	15	35	15	15	25	15
5		B2	15	20	10	10	15	15	15	10
6		C	20	0	10	40	35	30	50	20
7		NA	938	2180	945	278	1182	896	1329	1397
8		CAL	295	770	440	430	315	400	370	450
9										

図10-7　Excelにおける二次元のAMPLテーブル

2次元表の読込

```
table dietAmts IN "ODBC" "Diet2D.xls":
   [i ~ NUTR],
   {j in FOOD} <amt[i,j] ~ (j)>;
```

標準および組込みのテーブルハンドラ

10.7

- **標準 ODBC ハンドラ** — Access・Excel など ODBC ドライバを持つあらゆるアプリと接続（Windows）
- **組込みテキスト／バイナリ** — .tab（ASCII）／.bit（バイナリ）— デバッグ・実演用の最小実装
- **SQL クエリの埋め込み** — "SQL=SELECT …" で一時的な関係表を作ってから読み込める

SQL埋め込み例

```
table cheapFoods IN "ODBC" "diet.mdb"
   "SQL=SELECT * FROM Foods WHERE cost <= 2.49":
   FOOD <- [FOOD], cost, f_min, f_max;
```

✓ まとめ：第10章 データベースアクセス

1. table 宣言が、AMPLの集合・パラメータ・変数・式と、関係表の列との対応を定義する
2. 実際の入出力は read table / write table コマンドで実行される
3. キー列は <- ・ -> ・ <-> で、データ列は IN・OUT・INOUT で読み書きの方向を制御する
4. ~ 演算子とダミー添字により、列名の違い・番号のずれ・添字順序の違いに対応できる
5. 標準ODBCハンドラに加え、テキスト・バイナリの組込みハンドラやSQLクエリも利用できる

AMPL：数理計画のためのモデリング言語

第11章

モデリングコマンド

model・*solve* から *fix*・*drop*・*relax_integrality* まで ―
解いて、直して、また解く。対話的セッションを操るコマンド群

>_

全22章シリーズ　第11回

AGENDA

本章の内容

11.1 コマンドとオプションの一般原則

対話セッションの基本構造、option コマンドによる設定

11.2 モデルとデータの設定と求解

model / data / solve コマンド、目的関数の選択、ソルバー選択

11.3 データの変更

reset data / update data、再標本化、let コマンド、丸め関数

11.4 モデルの変更

delete / purge / redeclare、fix / unfix / drop / restore、整数性の緩和

コマンド環境の一般原則

▸ **対話的セッション**

ampl を起動すると ampl: プロンプトが現れ、入力されたコマンドを解釈・実行する

; **セミコロンで終端**

コマンドはセミコロンで終わる。1行に複数コマンドを書いても、複数行にまたがってもよい

? **継続プロンプト**

コマンドの途中で改行すると ampl? が現れ、続きの入力を促す

F **ファイル名**

セミコロンと引用符を除く印字可能文字の並び。規則は OS が決める

Q **end / quit**

セッションを終えるにはこのいずれかを打ち込む

amplsession

```
ampl: display {i in ORIG, j in DEST}
ampl? sum {p in PROD} Trans[i,j,p];

# ampl?  は継続入力を示す
```

コマンドはスクリプト（第13章）にも登場する。GUIのメニュー操作も裏ではこのコマンド体系を呼び出している。

option コマンド

- **各オプションは名前と値（数値または文字列）を持つ**
  - 例：prompt1・prompt2（文字列）、display_width（数値）
- **option 名前; で現在の値を表示**
- **option 名前 値; で値を再設定**
- **名前に * を使うとワイルドカード一致**
  - option だけ、または option *; で全オプション一覧
- **reset options; で既定値に一括復元**
  - AMPLは正しいオプション名の一覧を持たない — 打ち間違いは新しいオプションとして静かに定義されるので注意

オプションを見る

```
ampl: option prompt1, display_width;
option prompt1 'ampl: ';
option display_width 79;
```

オプションを設定

```
ampl: option prompt1 "A> ", display_width 60;
A> option display_width;
option display_width 60;
```

モデルとデータの設定と求解

diet.mod を解く

```
ampl: model diet.mod; data diet.dat; solve;
MINOS 5.5: optimal solution found.
6 iterations, objective 88.2
```

- model：モデル宣言（第5〜8章）を含むファイルを名指しし、現在のモデルに追加
- data：データ値（第9章）を含むファイルを名指しし、既存の構成要素に値を与える
- solve：最適化問題をソルバーへ送り、結果を取り出す
- 複数回の model / data コマンドで、異なるファイルから積み上げられる

M モデル / データ モード

- AMPL は常に「モデルモード」で開始
- ファイル名なしの data; でデータモードへ
- ファイル名なしの model; でモデルモードへ戻る
- option・solve・subject to 等もモデルモードへ復帰させる

Σ objective コマンド

単一の目的関数を選ぶ

```
objective Total_Number;
objective Total_Cost["A&P"];
```

solve コマンドの流れ

1

問題の生成

モデルとデータから具体的な最適化問題を作る。データ不足や制限違反はここでエラー

2

前処理（プリソルブ）

問題を解きやすく縮小。実行不可能を検出しソルバー送信を省くこともある

3

ソルバーへ送信

既定ソルバーは option solver で確認・変更できる

前処理が実行不可能を検出

```
ampl: model steel4.mod;
ampl: data steel4.dat;
ampl: let avail['reheat'] := 10;
ampl: solve;
presolve: constraint Time['reheat']
  cannot hold: body <= 10 cannot be
  >= 11.25; difference = -1.25
```

エラーの手がかり

- “ignoring integrality” → 整数計画対応ソルバーが必要
- “can't handle nonlinearities” → 非線形対応ソルバーが必要
- 解が遅い／メッセージなしで戻る → ソルバー固有オプションを確認

ソルバーの選択と比較

同じ問題を3つのソルバーで比較

```
ampl: model steelT.mod; data steelT.dat;
ampl: solve;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033

ampl: reset;
ampl: model steelT.mod; data steelT.dat;
ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal solution;
  objective 515033

ampl: reset;
ampl: model steelT.mod; data steelT.dat;
ampl: option solver snopt;
ampl: solve;
SNOPT 6.1-1: Optimal solution found.
15 iterations, objective 515033
```

使い方の基本

- option solver; で既定ソルバー名を確認
- option solver cplex; のように切り替え
- 多くの問題は solver 設定と solve だけで十分

なぜ毎回 reset するのか

reset なしだと、あるソルバーが見つけた解の情報が次に引き継がれ、比較が不公平になる。同じ初期条件から始めて性能を比較するために reset を挟む。

データの変更：reset data / update data

R reset data

- 指定した集合・パラメータの現在値を削除する
- 以後 solve すると未設定のためエラーになりうる

```ampl
reset data MINREQ, MAXREQ,
   amt, n_min, n_max;
```

U update data

- 新しい値が実際に読まれるまで、古い値は削除されない
- 読み込まれなかった項目は以前の値のまま残る

```ampl
update data MINREQ, MAXREQ,
   amt, n_min, n_max;
```

例：MINREQ・amt・n_min のみ新しい値を読み込んだ場合 — reset data なら MAXREQ・n_max は値を失いエラーになるが、update data なら以前の値のまま維持される。

乱数パラメータの再標本化

avail を正規乱数として定義

```
param avail_mean {STAGE} >= 0;
param avail_variance {STAGE} >= 0;
param avail {s in STAGE} =
   Normal(avail_mean[s], avail_variance[s]);
```

- reset data は乱数で計算されるパラメータを再標本化する
- 標本が変われば解も、最適目的値も変わる
- reset（データではなく全体）を出すと乱数生成器自体がリセットされ、値は最初から繰り返される

steel4r：再標本化のたびに解が変わる

```
ampl: solve;
MINOS 5.5: optimal solution found.
3 iterations, objective 187632.2489
ampl: display avail;
reheat 32.3504
  roll 43.038 ;

ampl: reset data avail;
ampl: solve;
MINOS 5.5: optimal solution found.
4 iterations, objective 158882.901
ampl: display avail;
reheat 32.0306
  roll 32.6855 ;
```

let コマンド

配合モデルの上限を試行錯誤で変更

```
ampl: let f_max["CHK"] := 11;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 73.43818182
```

- keyword let ＋（任意で）添字式 ＋ 代入 という形
- = で定義されていない集合・パラメータの値を直接変更できる
- 変数の現在値も変更でき、別の解を試すのに使える

添字集合をまとめて変更

```
let {j in FOOD} f_max[j] := 8;
let {j in FOOD}
   f_max[j] := 1.1 * f_max[j];
```

! 注意：添字づけへの影響

週数 T のような、他のデータの添字づけに使われる値を let で変えると、既存データの添字が無効になりエラーになりうる（Tdata と T を分けて宣言するのが定石）。

表11-1　丸め関数（抜粋）

`ceil(x)`	x の天井（次に大きい整数）
`floor(x)`	x の床（次に小さい整数）
`round(x, n)`	小数点以下 n 桁に丸め
`trunc(x)`	整数に切り捨て

モデル構成要素の除去と再定義

D

delete

他の構成要素から参照されていなければ取り除く。目的・制約はいつでも削除可

P

purge

指定要素と、それに直接・間接に依存するすべての構成要素をまとめて削除

X

xref

ある構成要素にどの構成要素が依存しているかを調べる

E

redeclare

宣言を改訂版で置き換える。依存要素やデータ値はそのまま保たれる

purge f_min の影響範囲を先に確認

```
ampl: xref f_min;
# 4 entities depend on f_min:
f_max
Buy
Total_Cost
Diet
```

redeclare の例

```
ampl: redeclare param f_min
   {FOOD} > 0 integer;

# f_min の妥当性条件だけを変更
# 依存要素・既存データ値は保持
```

fix・unfix・drop・restore

⊘ drop

特定の制約・目的を一時的に無視する

⊕ restore

drop の効果を打ち消し、制約を復活させる

F fix

変数を現在値に固定する（＝制約であるかのように扱う）

U unfix

fix の効果を打ち消す

いずれもキーワード＋任意の添字式＋名前という共通構文。drop/restore は制約・目的、fix/unfix は変数を対象とする。連続する drop の効果は累積する。

fix の実例（配合モデル）

```
ampl: let {j in FOOD} Buy[j] := f_min[j];
ampl: fix {j in FOOD: amt["NA",j]>1200}
        Buy[j];
ampl: solve;
MINOS 5.5: optimal solution found.
7 iterations, objective 86.92

# ナトリウム過多の食品を固定し
# 残りの変数だけを最適化
```

添字式・代入句も併用可能

```
drop {i in MAXNOT} Diet_Max[i];
fix {j in FOOD:...} Buy[j] := f_min[j];
```

整数性の緩和

0/1 relax_integrality オプション

```ampl
option relax_integrality 1;
```

- 既定 0 → 1 にすると整数制限をすべて無視
- integer 変数は元の上下限のまま連続値に
- binary 変数は下限0・上限1の連続値に

.r .relax 接尾辞

```
let {j in FOOD: f_min[j]>allow_frac}
   Buy[j].relax := 1;
```

AMPL の緩和 vs ソルバーの緩和指示

scale=0.95、f_max=10 の配合モデル（整数変数）で比較：

緩和なし

objective 122.89

上限9.5を整数条件で9に前処理で切り下げ

ソルバーの relax 指示

objective 120.24

前処理は9のまま。ソルバー側だけ緩和

relax_integrality 1

objective 119.15

前処理も9.5のまま。最も緩い解

SUMMARY

第11章のまとめ

1	**対話環境の基本**	ampl: プロンプト、セミコロン終端、option コマンドによる振る舞いの制御
2	**model・data・solve**	現在のモデルを積み上げ、objective で目的を選び、solver でソルバーを切り替える
3	**データの再設定**	reset data／update data で値を入れ替え、let で個別値をピンポイントに変更
4	**モデルの限定変更**	delete・purge・redeclare で構成要素を、fix・drop で一時的な制約緩和を行う
5	**整数性の制御**	relax_integrality と .relax 接尾辞で、必要な変数だけ整数条件を緩められる

次章：第12章へ続く

AMPL：数理計画のためのモデリング言語

第12章

表示コマンド

display・print・printf で結果を読み解き、思いどおりに整形する

集合・値の表示 | 書式と数値オプション | 解の双対値・被約費用 | モデル検査 (show/xref/expand)

Fourer, Gay, Kernighan 著（日本語版）

AGENDA

本章の内容

12章

12.1 結果を眺める：display コマンド

12.2 display の書式オプション

12.3 display の数値オプション

12.4 その他の出力コマンド：print と printf

12.5 解に関連する値

12.6 モデルと問題例に関するその他の表示機能

12.7 出力を操作する一般的な機能

結果を眺める：display コマンド

基本

- display に見たいものの記述を続けて打ち込むだけで、値を見慣れた配置に自動整形する。
- 第9章のデータ文と同じリスト・表の形式を可能なかぎり使う。
- 集合の内容は display と集合名のリストで表示できる。
- 添字づけられた集まりの各集合も一括表示できる（例：PROD, AREA）。
- 引数は宣言済みの集合でなくてよく、集合に評価される任意の式を渡せる。

対話例（*amplsession*）

```
ampl: display ORIG, DEST, LINKS;
set ORIG := GARY CLEV PITT;
set DEST := FRA DET LAN WIN STL FRE LAF;
set LINKS :=
(GARY,DET) (GARY,LAF) (CLEV,LAN) ...
(PITT,STL) (CLEV,FRA) (CLEV,WIN) ...;

ampl: display union {p in PROD} AREA[p];
set union {p in PROD} AREA[p] :=
   east north west export;
```

パラメータと変数の表示

次元別

1 1次元データ

単純な集合で添字づけられたパラメータ・変数は列（リスト）形式。例：avail

2 2次元データ

対の集合・二つの集合で添字づけられた値。少なければリスト、多ければ表になる。

3+ 3次元以上

AMPLは二次元の表に「スライス」して提示する。表の数が最小になる軸を自動選択。

Σ スカラー・式

display T や display sin(1)^2+cos(1)^2 のように単一値もそのまま表示できる。

対話例：1次元→2次元→3次元(スライス)

```
ampl: display avail;
avail [*] :=
reheat 35
  roll 40 ;

ampl: display supply;
supply :=
CLEV bands  700
CLEV coils 1600
GARY bands  400  ... ;

ampl: display Trans;
Trans [CLEV,*,*]
: bands coils plate :=
DET   0   750    0  ...
```

添字づけられた式の表示

式の表示

- 任意の算術式を display できるが、配列同士の演算（revenue * Sell）は認識されない。
- 式の添字づけられた集まりを表示するには、添字づけを明示的に指定する。
- 添字式＋(式のリスト) は1つの「表示項目」として扱われる。
- display コマンド全体を添字づければ、添字集合の各要素ごとに個別出力できる（表のスライスの並べ替えに有用）。

対話例（誤り例→正しい書き方）

```
ampl: display revenue * Sell;
syntax error

ampl: display {p in PROD, t in 1..T}
ampl?    revenue[p,t]*Sell[p,t];
revenue[p,t]*Sell[p,t] [*,*] (tr)
:   bands    coils :=
1   150000    9210
2   156000   87500 ...

ampl: display {p in PROD}:
ampl?   {i in ORIG,j in DEST} Trans[i,j,p];
Trans[i,j,'bands'] [*,*] (tr) ...
```

display の書式オプション（表12-1）

表12-1

オプション	意味（既定値）
`display_1col`	表をリスト形式で表示する要素数の上限（20）
`display_transpose`	行数 − 列数 ＜ display_transpose なら表を転置（0）
`display_width`	1行あたりの最大文字数（79）
`gutter_width`	表の列間の空白幅（3）
`omit_zero_cols`	非0なら、全て零の列を表示から省く（0）
`omit_zero_rows`	非0なら、全て零の行を表示から省く（0）

リストと表の配置・転置

配置

display_1col：0 でコンパクト表示を強制

```
ampl: option display_1col 0;
ampl: display required;
required [*] :=
Fri1 100  Mon1 100  Sat1 100 ...
Fri2  78  Mon2  78  Sat2  78 ...
Fri3  52  Mon3  52  Thu1 100 ...;
```

display_transpose：正=転置優勢／負=抑制

```
ampl: option display_transpose -5;
ampl: display cost;
cost [*,*]
:     DET FRA FRE LAF LAN STL WIN :=
CLEV    9  27   .   17  12  26   9
GARY   14   .   .    8  11  16   .
PITT    .  24  99    .   .  28  13 ;
```

- display_1col：表示値の個数がこの値以下なら1列リスト、超えればコンパクトな形式（表）を使う。既定20。
- 表の向きは既定で「行 ＞ 列」を優先。列数が行数を超える場合は転置され、先頭行に (tr) が付く。
- display_transpose：規則は「行数 − 列数 ＜ display_transpose のときのみ転置」。

行幅の制御と零の抑制

幅と零

行幅の制御

- display_width：1行の最大文字数（既定79）。超える表は縦に2つ以上の表へ分割される。
- 列見出しが値よりずっと広い場合は $1, $2 のような略号が自動導入される。
- 見出しが値より狭ければ gutter_width（列間の空白、既定3）を減らして詰められる。

零の抑制

- omit_zero_rows を1にすると、全て零の行が表示から省かれる。
- omit_zero_cols を1にすると、全て零の列が表示から省かれる。
- 割当問題のようにほぼ0の変数が多い場合に、関心のある項目だけに絞り込める。

対話例：零を含む大きな表を圧縮する

```
ampl: option omit_zero_rows 1;
ampl: display Trans;
Trans :=
Coullard C118  1
Daskin   D241  1
Hazen    C246  1  ... ;
```

display の数値オプション（表12-2）

表12-2

オプション	意味（既定値）
`display_eps`	零と区別して表示する最小絶対値（0）
`display_precision`	表示数値を丸める有効桁数。0なら完全精度（6）
`display_round`	小数点位置での丸め桁。display_precision に優先（空）
`solution_precision`	解の値そのものを丸める有効桁数（0）
`solution_round`	解の値そのものを丸める桁位置。solution_precision に優先（空）

見え方だけを変える： display_eps / display_precision / display_round　　**実際の値を変える：** solution_precision / solution_round（solve の前に設定）

数値の見え方と丸め

丸め

precision（有効桁数）と round（丸め位置）

```
ampl: option display_precision 3;
Make['bands',4]/rate['bands'] = 32.1

ampl: option display_precision 0;
...  = 32.14285714285713

ampl: option display_round 2;
:  bands  coils :=
1  29.95  10.05
4  32.14   7.86 ;
```

- display_precision：既定6桁。0なら計算機表現から必要十分な桁数で正確に表す。
- display_round：小数点以下の桁数を固定（非負）／百・十の位に丸め（負）。整数値なら precision に優先。
- display_eps：この絶対値未満は厳密な零として扱い、1e-17 のような微小残差の表示を防ぐ。
- **注意：これらは見え方のみを変え、solve 後の実際の解の値には影響しない。**

その他の出力コマンド：print と printf

整形出力

print：整形しない1行出力

```
ampl: print {p in PROD} (p, rate[p]);
bands 200 coils 140

ampl: print {t in 1..T}: {p in PROD} Make[p,t];
5990 1407
6000 1400 ...
```

- 集合は引数にできないが、その要素は渡せる。添字づけを入れ子にできる点が display と異なる。
- print_separator でスペース以外の区切り（タブ等）を指定でき、表計算への出力に便利。

printf：C言語風の書式指定

```
ampl: printf "Total revenue is $%6.2f.\n",
ampl?    sum {p in PROD,t in 1..T}
ampl?       revenue[p,t]*Sell[p,t];
Total revenue is $787810.00.
```

- 最初の印字項目が書式文字列。%6.2f のような変換指定で幅・小数点以下の桁数を制御する。
- 改行は \n で明示。添字式＋コロンを使えば要素ごとに1行を自動生成できる。
- 変換指定の数と値の数が一致する必要があり、列数が変わる表の出力には不向き。

目的関数・上下限・スラック

接尾辞

- AMPLは「変数名／制約名 ＋ ドット ＋ 接尾辞」の修飾名で、最適解に付随する量を参照する。
- 目的関数の名前はそのまま最適値を指す（display、print、printf で利用可）。
- .lb / .ub 上下限（AMPLの前処理が導いた上下限を含む）
- .slack 値と近いほうの限界との差
- .body 制約本体（下限 ≦ 本体 ≦ 上限）の現在値

対話例：上下限・本体・スラックの表示

```
ampl: display Buy.lb, Buy, Buy.ub, Buy.slack;
:    Buy.lb   Buy   Buy.ub Buy.slack :=
BEEF    2      2       10       0
CHK     2     10       10       0
MTL     2   6.236      10   3.764
SPG     2   5.258      10   3.258 ;

ampl: display Diet_Min.lb, Diet_Min.body,
ampl?   Diet_Min.ub;
:   .lb    .body     .ub :=
A   700  1013.98  Infinity
C   700   700.00  Infinity ;
```

双対値と被約費用

感度

- 制約の双対値（潜在価格）：修飾接尾辞なしの制約名で参照。制約の右辺（限界）が1単位変化したときの目的値の変化率を表す。
- スラックが正の制約では双対値は零。スラックが零のとき、双対値は意味をもつ。
- 変数側の対応物は被約費用（.rc）。上下限のいずれかで値が止まっている変数で非零になりうる。
- 目的値を制約の定数項に対して図示すると、区分的線形の曲線になる（右図）。双対値・被約費用はその曲線の傾き＝局所的な感度を1つだけ与える。

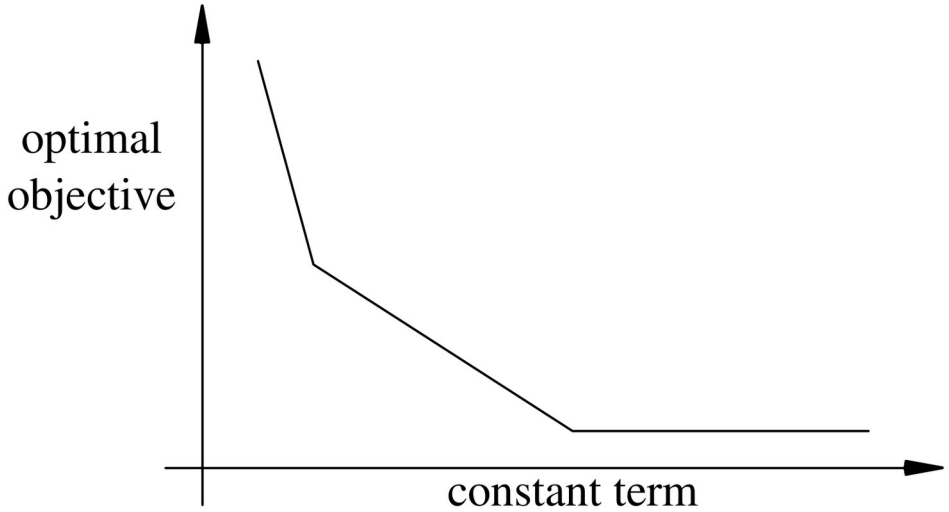


図12-1：目的関数の区分的線形なグラフ
縦軸＝最適な目的値、横軸＝制約の定数項（限界）

モデル構成要素と問題例の検査

検査コマンド

show — モデル構成要素の一覧・宣言の表示

単独で全構成要素名を種類別に一覧。show Total_Cost のように個別の宣言も表示でき、モデルファイルを探す手間を省く。

xref — 依存関係の表示

指定した構成要素に直接・間接に依存するすべての構成要素（制約・目的など）を一覧する。

expand — 生成された問題例の表示

モデルとデータから実際に生成された制約・目的を展開表示。solexpand は前処理後の「ソルバーの視点」を示す。

対話例：show / xref / expand

```
ampl: show;
parameters: demand fcost limit ...
variables:  Trans  Use
constraints: Demand Max_Serve ...

ampl: xref demand, Trans;
# 2 entities depend on demand:
check 1
Demand

ampl: expand Supply;
subject to Supply['GARY']:
  Trans['GARY','FRA'] + ... = 1400;
```

総称的な別名と資源の一覧

総称名

変数・制約・目的の総称的な別名

- _nvars / _ncons / _nobjs：現在の問題の変数・制約・目的の数。
- _varname[1.._nvars] など：各構成要素の真のAMPL名を返す文字列パラメータ。
- _var[j] / _con[j] / _obj[j]：番号で構成要素を参照する別名。_var[5].rc のように接尾辞も使える。
- 接頭辞を _s に置き換えると（_svar 等）、前処理後の「ソルバーの視点」を参照できる。

資源の一覧

- show_stats：solve 時に問題の規模（変数・制約数、非零要素数など）を要約表示。
- times：AMPL翻訳系の所要時間とメモリの要約を表示。
- gentimes：genmod 段階の詳細な資源使用量。処理が遅い原因の特定に有用。
- _solve_time / _ampl_time：直近の solve に要したCPU秒数を保持する定義済みパラメータ。

対話例：上限に達していない変数だけを列挙

```
ampl: display {j in 1.._nvars:
ampl?  _var[j] < _var[j].ub - 0.00001} _varname[j];
_varname[j] [*] :=
2 "PD_Ship['SE']"          5 "DW_Ship['NE','BWI']"
6 "DW_Ship['SE','EWR']"    7 "DW_Ship['SE','BWI']" ;
```

出力を操作する一般的な機能

出力制御

リダイレクト（>／>>）

display / print / expand などのコマンド末尾に >filename を付ければ結果をファイルに保存。>> は追記、close で明示的に閉じる。

log_file オプション

以降のコマンドと応答をまとめて記録。log_model / log_data で入力ファイルの内容も記録に含められる。

メッセージの制限

eexit（既定-10）でエラー時の打ち切り方を制御。presolve_warnings（既定5）、bad_subscripts（既定3）で警告数を制限。

対話例：リダイレクト／記録／打ち切り

```
ampl: display supply >multi.out;
ampl: close multi.out;

ampl: option log_file 'multi.tmp';

ampl: option eexit -3;
ampl: model diet.mod;
ampl: data diet.mod;
diet.mod, line 4 (offset 32):
  expected ; ( [ : or symbol
Bailing out after 3 warnings.
```

SUMMARY

✓ 本章のまとめ

1	**display**	集合・パラメータ・変数・任意の式を、1次元はリスト、2次元以上は表やスライスへ自動整形して表示する。
2	**書式・数値オプション**	display_1col / display_transpose / display_width / omit_zero_* で配置を、display_precision / display_round / display_eps で見え方を制御する。solution_* は実際の値を丸める。
3	**print と printf**	整形しない出力と、C言語風の書式で完全に制御した出力を使い分け、他ツール向けデータや報告書を作る。
4	**解に付随する値**	.lb / .ub / .slack / .body、制約の双対値、変数の被約費用（.rc）で最適解を深く解釈できる。
5	**モデル検査と出力制御**	show / xref / expand でモデルと問題例を検査し、> によるリダイレクトや log_file、eexit 等で出力とメッセージ量を管理する。

次章：第13章へ続く

AMPL：数理計画のためのモデリング言語

>_

第13章　コマンドスクリプト

include・*for*・*repeat*・*if* で解析作業を自動化する

`include / commands` `for` `repeat` `if-then-else` `break / continue` 文字列関数

本章の内容

AGENDA

13.1	スクリプトの実行：include と commands	ファイルからコマンド列を読み込む2つの方法
13.2	集合上の反復：for 文	集合の各要素について文を繰り返す
13.3	条件付きの反復：repeat 文	while / until で終了条件を指定する
13.4	条件の検査：if-then-else 文	条件に応じて実行する文を切り替える
13.5	ループの終了：break と continue	回をスキップ／ループを打ち切る
13.6	スクリプトを一歩ずつ実行する	single_step・step・next・skip でデバッグ
13.7	文字列の操作	連結演算子＆と文字列関数、文字列式の活用

なぜコマンドスクリプトか

- モデル開発の初期段階では対話的にコマンドを打ち込むが、定式化が落ち着くと
- **同じコマンド列を何度も繰り返し入力することになる**
- **スクリプト（script）＝ファイルに収めた、繰り返し使うためのコマンド列**
- for、repeat、if など、プログラミング言語のような制御構文を含められる
- → AMPL のコマンド言語で「小さなプログラム」が書ける
- 単発実行 include／commands から、繰り返し for／repeat、条件分岐 if、
- 文字列処理まで、この章で段階的に扱う

1. **読む**
   `model / data / include`
2. **繰り返す**
   `for / repeat`
3. **分岐する**
   `if-then-else`
4. **制御する**
   `break / continue`
5. **調べる**
   `single_step / step`

IN スクリプトの実行：include と commands

`include filename`

- 指定ファイルの内容で置き換えられる
- 他の文の途中にも現れてよい
- 終端のセミコロンは不要
- モードを変えない（model/dataの一般形）

`commands filename ;`

- include に似ているが、独立した「文」である
- 終端のセミコロンが必要
- 文が合法な文脈にのみ現れうる
- **for / repeat / if 内では commands が最善**

- model・data コマンドは、読み込み前にモードをモデル／データモードにする include の特別な場合
- 小さなモデルでは、宣言・data コマンド・データ文を1ファイルにまとめ model や include で一度に読み込める
- 起動時に option の既定値を変える include ファイルを自動で読み込ませることもできる（付録 A-14, A-23）

感度分析スクリプトを commands で実行

steelT.sa1 （繰り返すコマンドを1ファイルに）

```
solve;
display Total_Profit >steelT.sens;
option display_1col 0;
option omit_zero_rows 0;
display Make >steelT.sens;
display Sell >steelT.sens;
option display_1col 20;
option omit_zero_rows 1;
display Inv >steelT.sens;
let avail[3] := avail[3] + 5;
```

対話セッション：呼ぶたびに avail[3] が5増える

```
ampl: model steelT.mod;
ampl: data steelT.dat;
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
1 iterations, objective 532033
ampl: commands steelT.sa1;
MINOS 5.5: optimal solution found.
1 iterations, objective 549033
```

- commands steelT.sa1 の1行を打ち込むだけで、ファイル中の全コマンドが実行される
- display の出力はすべてファイル steelT.sens にリダイレクトされている（画面表示も可能）
- 同じ内容でも include で代用できるが、制御構文と組み合わせるスクリプトでは commands が安全

集合上の反復：for 文


```
for {添字式} コマンド;
for {添字式} { コマンド1; コマンド2; ... }
```

- ある集合の各要素について一度ずつ、文（または波括弧でまとめた複数の文）を実行する
- for と本体の間の式は任意の AMPL 添字式でよい
- コマンドを別ファイルに置いて commands で呼ぶ代わりに、波括弧内に直接書ける
- 集合が順序づけられていれば（1..T や ordered/circular 宣言）その順序で反復する
- 順序づけられていない集合では AMPL が順序を選ぶが、毎回同じ順序になる

1回の for で4回の感度分析を実行

```
ampl: model steelT.mod;
ampl: data steelT.dat;
ampl: for {1..4} commands
         steelT.sa1;
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
MINOS 5.5: optimal solution found.
1 iterations, objective 532033
MINOS 5.5: optimal solution found.
1 iterations, objective 549033
MINOS 5.5: optimal solution found.
2 iterations, objective 565193
```

FOR パラメータ感度分析のスクリプト

図13-1

steelT.sa3

```
set AVAIL3;
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};

let AVAIL3 := avail[3] .. avail[3]+15 by 5;

for {a in AVAIL3} {
   let avail[3] := a;
   solve;
   let avail3_obj[a] := Total_Profit;
   let avail3_dual[a] := Time[3].dual;
}

display avail3_obj, avail3_dual;
```

```
ampl: commands steelT.sa3;
: avail3_obj avail3_dual :=
32   515033      3400
37   532033      3400
42   549033      3400
47   565193      2980
;
```

- AVAIL3 は試したい avail[3] の値の集合
- for {a in AVAIL3} が各値 a について解き直す
- 添字 a はループ内の文でそのまま使える
- option solver_msg 0 でソルバーの出力を抑制

FOR 入れ子ループで整形表を作る

図13-2

steelT.tab1 （外側 1..T、内側 PROD の二重ループ）

```
printf "\nSALES";
printf {p in PROD}: "%14s   ", p;
printf "\n";

for {t in 1..T} {
   printf "week %d", t;
   for {p in PROD} {
      printf "%9d", Sell[p,t];
      printf "%7.1f%%", 100*Sell[p,t]/market[p,t];
   }
   printf "\n";
}
```

```
ampl: commands steelT.tab1;
SALES         bands     coils
week 1     6000 100.0%   307   7.7%
week 2     6000 100.0%  2500 100.0%
week 3     1399  35.0%  3500 100.0%
```

- 外側ループ {1..T} が表の1行を生成
- 内側ループ {p in PROD} がその行の各製品の項目を生成
- printf は改行 \n が現れるまで行を変えない
- 製品数や名前がデータ依存でも対応できる（1つの display文より汎用的）

REP 条件付きの反復：repeat 文

- 論理条件が満たされるかぎり反復を続ける
- while：条件が真であるかぎり継続
- until：条件が偽であるかぎり継続（真になったら終了）
- 条件なしなら無限ループ（break 等で終了）

`repeat while COND { ... }`	本体の前に検査 → 最初から偽なら0回
`repeat until COND { ... }`	本体の前に検査 → 最初から真なら0回
`repeat { ... } while COND`	本体の後に検査 → 少なくとも1回実行
`repeat { ... } until COND`	本体の後に検査 → 少なくとも1回実行

REP 感度を記録する repeat スクリプト

図13-3

steelT.sa4

```
set AVAIL3 default {};
param avail3_obj {AVAIL3};
param avail3_dual {AVAIL3};
param avail3_step := 5;

repeat {
   solve;
   let AVAIL3 := AVAIL3 union {avail[3]};
   let avail3_obj[avail[3]] := Total_Profit;
   let avail3_dual[avail[3]] := Time[3].dual;
   let avail[3] := avail[3] + avail3_step;
} until Time[3].dual = 0;

display avail3_obj, avail3_dual;
```

- 反復回数が事前に分からないので AVAIL3 を default {} で空集合に
- **= {} ではなく default {} が重要：**
- **let で以後も変更可能にするため**
- 解いた後で新しい avail[3] を集合に追加していく
- 浮動小数点誤差対策として option solution_precision 10 を設定
- 本体の後の until … なので少なくとも1回は solve される

条件の検査：if-then-else 文


```
if Make["coils",2] < 1500 then
   printf "under 1500\n";

if Make["coils",2] < 1500 then {
   printf "Fewer than 1500 coils.\n";
   let market["coils",2] :=
      market["coils",2] * 1.1;
}
else
   printf "At least 1500 coils.\n";

// else if の連鎖：
if Sell[p,t] = 0 then printf "    ";
else if Sell[p,t] < market[p,t] then
   printf "%7.1f%%", 100*Sell[p,t]/market[p,t];
else printf "  --- ";
```

- if 条件式 then 動作; が最も単純な形
- 動作は単一コマンドでも { … } でまとめた複数コマンドでもよい
- 任意の else が代替の動作を指定する（単一コマンドまたはブロック）
- 式が真なら then 側、偽なら else 側（あれば）を実行
- then／else の中身がさらに if 文でもよく、else if を連ねて多分岐にできる
- else は直前の対応可能な if と対にされる

BRK ループの終了：break と continue

`continue`

現在の回を終了し、次の回の検査から実行を続ける

`break`

for／repeat ループを完全に終了し、直後の文へ制御を移す

名前付きループ

repeat name { … } と break name; で外側ループを直接指定

steelT.sa7（図13-5）：変化があった時だけ記録

```
repeat {
   let avail[3] := avail[3] + 1;
   solve;
   if Time[3].dual = previous_dual
      then continue;
   let AVAIL3 := AVAIL3 union {avail[3]};
   ...
   if Time[3].dual = 0 then break;
   let previous_dual := Time[3].dual;
}
```

- break・continue は最も内側のループにのみ適用される
- 外側のループを直接抜けたい場合はループに名前を付ける：
- repeat sens_loop { … }
- for {t in 1..T}
- if Time[t].dual<2700 then
- break sens_loop;
- ループ名は repeat／for の直後、break／continue の直後に置く

スクリプトを一歩ずつ実行する

- option single_step 1; で1コマンドずつ止まる「シンボリックデバッガ」に
- **step / step n**
- 次の1個／n個のコマンドを実行
- **next / next n**
- 複合コマンド（for/repeat/if）の中に入らず、その先へ進む
- **skip / skip n**
- 次の1個／n個を実行せずに飛ばす
- 止まった位置は ファイル名:行番号(文字番号) で示される
- 一歩ずつ実行中も任意の AMPL コマンドを打ち込める（display 等で状態確認）
- <n>ampl: プロンプトはこのモードにいることを示す

対話セッション：<2>ampl は入れ子水準を示す

```
ampl: option single_step 1;
ampl: commands steelT.sa7;
steelT.sa7:2(18)   data ...
<2>ampl: step
steelT.sa7:4(36)   option ...
<2>ampl: step
steelT.sa7:5(66)   option ...
<2>ampl: display avail[3],
   Time[3].dual, previous_dual;
avail[3] = 22
Time[3].dual = 3620
previous_dual = 3620
<2>ampl: step
steelT.sa7:17(317)   continue ...
```

& 文字列の操作：関数と演算子

`&`	連結演算子。2つの文字列（数値は自動変換）をつなげる	`"WEEK" & t → "WEEK3"`
`length(s)`	文字列 s の文字数を返す	`length("FISH") = 4`
`match(s,t)`	s の中で t が最初に現れる位置。なければ0	`match("CHK","H") = 2`
`substr(s,p,n)`	位置 p から始まる長さ n の部分文字列	`substr("A_BEEF",1,1) = "A"`
`sub / gsub`	正規表現 t の最初／すべての出現を置換	第2引数は正規表現として解釈
`sprintf(...)`	printf と同じ書式変換で文字列を返す（出力しない）	`sprintf("WEEK%3.1f",0.3)`

& 文字列式でファイル名・オプションを組み立てる

13.7

- model, data, commands のファイル名、リダイレクト先（>, >>）、
- option の値、table の string-list などに文字列式が使える
- いずれの場合も文字列式は括弧で囲む
- 添字 j や i, j はループ中で文字列に自動変換される
- ループを通るたびに異なるファイル・異なるオプション値を設定できる

diet1.dat〜diet3.dat を順に解き diet1.out〜.out へ保存

```
model diet.mod;
set CASES = 1 .. 3;

for {j in CASES} {
   reset data;
   data ("diet" & j & ".dat");
   solve;
   display Buy >("diet" & j & ".out");
}
```

組み合わせを変えて4通りの CPLEX 指示で解く

```
model sched.mod;  data sched.dat;
option solver cplex;

set DIR1 = {"primal","dual"};
set DIR2 = {"primalopt","dualopt"};

for {i in DIR1, j in DIR2} {
   option cplex_options (i & " " & j);
   solve;
}
```

まとめ

1. **スクリプトはファイルに収めたコマンド列**

   include はどこにでも現れる置換、commands は独立した文。制御構文と併用するなら commands が安全

2. **for 文で集合上を反復**

   任意の添字式・任意の集合上で反復でき、入れ子にして printf と組み合わせれば整形表も作れる

3. **repeat 文で条件付き反復**

   while／until を本体の前後どちらに置くかで、最低実行回数（0回 or 1回）が決まる

4. **if-then-else と break／continue でループ制御**

   多分岐や条件付きスキップ・終了、名前付きループで外側ループも直接制御できる

5. **文字列演算子 & と関数、single_step デバッグ**

   文字列式でファイル名・オプション値を動的に組み立て、step/next/skip で挙動を検証できる

AMPL：数理計画のためのモデリング言語



第14章

ソルバーとのやりとり

前処理・求解結果・接尾辞・複数モデルの交互求解までを、*AMPL*とソルバーの対話として理解する

AGENDA

本章の内容

14.1 前処理	ソルバーへ送る前に問題を簡約する仕組みと制御オプション
14.2 求解結果の取り出し	solve_result・変数状態(.sstatus)・AMPL状態(.astatus)
14.3 接尾辞による情報交換	ユーザー定義／ソルバー定義の接尾辞でソルバーと対話
14.4 モデルを交互に使う	列生成・分解などのアルゴリズム的手続きの実装
14.5 名前付き問題	problem文・environ文でモデルの一部を切り替える

前処理(presolve)の働き

- solve コマンドの実行時、問題例がソルバーに送られる前に自動的に実行される
- 変数を1つしか含まない制約を検出し、変数を固定または上下限に畳み込んで除去
- 現在の上下限から冗長と証明できる制約を検出して除去
- 簡約はすべて解が返された後に元に戻されるので、解はもとの問題の用語で見られる
- **オプション show_stats を1にすると効果がレポートされる**

```
ampl: model steelT.mod; data steelT.dat;
ampl: option show_stats 1;
ampl: solve;
Presolve eliminates 2 constraints
   and 2 variables.
Adjusted problem:
24 variables, all linear
12 constraints, all linear; 38 nonzeros
MINOS 5.5: optimal solution found.
15 iterations, objective 515033
```

図*4-4 steelT.mod の求解例：*2制約・2変数が前処理で除去される

除去された構成要素の確認: 状態(.status)が "pre" の変数・制約を print で列挙できる

前処理の効果を制御するオプション

`presolve`	0で前処理を完全に停止、1で第2部のみ停止。既定は10(走査回数の上限)
`var_bounds`	既定1は狭い上下限、2にすると第2の上下限の組をソルバーへ渡す
`presolve_inteps`	整数変数の上下限を丸める際の許容誤差。既定 1.0e-6
`presolve_eps`	実行不可能性の検出を緩和する許容誤差。既定 0
`presolve_fixeps`	ほぼ等しい上下限をもつ変数・制約本体を平均値に固定する閾値
`presolve_warnings`	表示する前処理メッセージ数の上限。既定 5

接尾辞 `.lb0/.ub0`(原上下限)、`.lb1/.ub1`(第1組)、`.lb2/.ub2`(第2組)、`.lb/.ub`(現在送られる値) で各段階の上下限を調べられる

前処理による実行不可能性の検出

```
ampl: let market["bands"] := 500;
ampl: solve;
inconsistent bounds for var
  Make['bands']:
  lower bound = 1000
  > upper bound = 500;
  difference = 500
```

単純な場合：変数の上限が下限を下回る

```
ampl: let avail := 13;
ampl: solve;
presolve: constraint Time
  cannot hold:
  body <= 13 cannot be
  >= 13.2589;
  difference = -0.258929
```

高度な検査：複数変数にまたがる矛盾も検出

- presolve_eps を小さな正の値に設定すると、丸め誤差による見かけ上の矛盾を無視できる（既定0、上限 presolve_epsmax=1.0e-5）

solve_result と solve_result_num

solve_result_num	solve_result	解釈
0-99	solved	最適解が見つかった
100-199	solved?	最適解が示されたが誤りの可能性あり
200-299	infeasible	制約を満たせない
300-399	unbounded	目的を限りなく改善できる
400-499	limit	設定した限界(反復回数等)で停止
500-599	failure	ソルバー内のエラー条件で停止

```
ampl: display
   solve_result_num, solve_result;
solve_result_num = -1
solve_result = '?'

ampl: solve;
MINOS 5.5: infeasible problem.

ampl: display
   solve_result_num, solve_result;
solve_result_num = 200
solve_result = infeasible
```

スクリプトでは *solve_result* を検査して分岐するのが定石*(200<=n<300* より簡潔*)*

応用例：実行不可能性を検査する感度分析

```
# diet.run（図14-1・抜粋）
option solver_msg 0;

for {i in nstart .. 0 by -nstep} {
   let n_max[N] := i;
   solve;
   if solve_result = "infeasible"
      then {
      printf "--- infeasible at %d ---\n", i;
      break;
   }
   let N_MAX := N_MAX union {i};
   let N_obj[i] := Total_Cost;
   let N_dual[i] := Diet[N].dual;
}
display N_obj, N_dual;
```

- 栄養素の上限を刻み幅ずつ減らしながら solve を繰り返す
- if solve_result = "infeasible" で実行不可能を検出し break
- 実行可能な間は目的値と双対値を記録して感度を分析
- solver_msg を0にすると毎回のソルバーメッセージを抑制できる

```
--- infeasible at 48000 ---
:     N_obj      N_dual    :=
51000 115.625   -0.0021977
60000 101.013    7.04e-19
```

ソルバー起動の失敗を検出する

- **solve_exitcode：ソルバー起動の成否を記録**
- 初期値 -1、起動成功で 0、失敗時はシステム依存の非零値
- よくある失敗原因：ソルバー名の綴り間違い、ライセンス不備、一時ファイルの書き込み失敗
- solve_exitcode_max（既定0）を超えると、include・for・repeat等の複合文を中断
- solve_message：最新のソルバー返却メッセージを保持する記号パラメータ
- solver_msg を0にすればメッセージ表示自体を抑制できる

```
ampl: option solver xplex;
ampl: solve;
Cannot invoke xplex:
  No such file or directory

ampl: display solve_exitcode;
solve_exitcode = 1024

ampl: display solve_result,
   solve_result_num;
solve_result = '?'
solve_result_num = -1
```

変数のソルバー状態 .sstatus

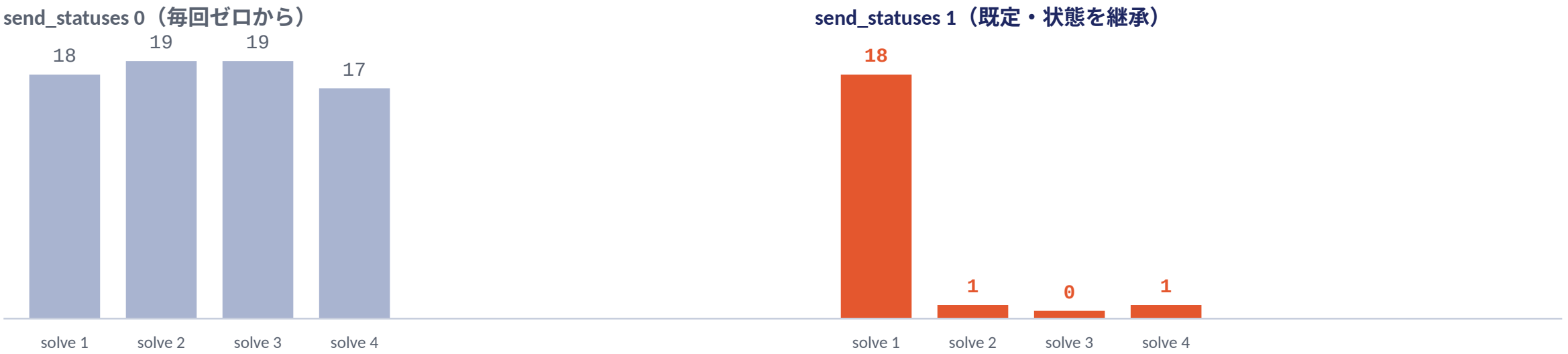
```
ampl: solve;
MINOS 5.5: optimal solution found.

ampl: display Buy.sstatus;
Buy.sstatus [*] :=
BEEF bas    CHK  low
FISH low    HAM  upp
MCH  upp    MTL  upp
SPG  bas    TUR  low
;
```

sstatus_table の値

bas	基底 (basic)
low	下限で非基底
upp	上限で非基底
sup	超基底（非線形ソルバー）
equ	上下限が等しい状態で非基底
none	状態未割当

send_statuses による再最適化の高速化

- send_statuses（既定1）は、各 solve で変数の基底状態をソルバーへ送り、次回の出発点に使う
- 労働力を5%ずつ増やして繰り返し解く例：状態を送るだけで反復回数が激減する



単体法の反復回数（*steelT3.mod*, *CPLEX*）

制約の状態と AMPL 状態

制約のソルバー状態 .sstatus

- 各制約に付随する「論理変数」(スラック/人工変数)の基底状態を表す
- 不等式制約はスラック変数を持ち bas/low/upp などをとる
- 等式制約は人工変数を持ち、非基底時は equ

AMPL 状態 .astatus

- in：問題に含まれる／drop：drop コマンドで除去
- pre：前処理で除去／fix：fix コマンドで固定
- sub：代入で除去／unused：目的・制約に未使用

.status = 統合されたひとつの状態

name.status は、name.astatus が "in" なら name.sstatus に等しく、そうでなければ name.astatus に等しい。通常はどちらか一方の状態にしか関心がないため、この統合接尾辞が便利。

ユーザー定義の接尾辞：整数計画の指示

- **suffix 文でソルバー固有の接尾辞を宣言する**
- IN：AMPLが書き出しソルバーが読む値（例：分枝優先度）
- CPLEXの .priority（0-9999）と .direction（-1,0,1）で
- 分枝の優先順位と方向を変数ごとに指定できる
- let 文や var 宣言中の suffix 句で値を割り当てる

```
ampl: suffix priority IN, integer,
ampl?    >= 0, <= 9999;
ampl: suffix direction IN, integer,
ampl?    >= -1, <= 1;

ampl: let {i in ORIG, j in DEST}
ampl?    Use[i,j].priority :=
ampl?    sum {p in PROD} demand[j,p];
ampl: let Use["GARY","FRE"]
ampl?    .direction := -1;

ampl: solve;
CPLEX 8.0.0: optimal integer
  solution; objective 235625
799 MIP simplex iterations
69 branch-and-bound nodes
```

ソルバー定義の接尾辞：感度分析

cplex_options に `'sensitivity'` を指定すると、CPLEXが .down / .current / .up の3接尾辞を自動生成する（OUT：ソルバーが書き出す）

```
ampl: option cplex_options 'sensitivity';
ampl: solve;
...suffix current OUT; (自動実行)

ampl: display Sell.down, Sell.current,
ampl?    Sell.up;
:      Sell.down Sell.current Sell.up :=
bands1  23.3      25          1e+20
bands3  24.9      27          27.5
coils2  33        35          1e+20
;
```

意味

- 変数：.current は現在の目的係数、.down/.up は現在の基底が最適のままである係数の範囲
- 制約：右辺値(定数項)に対する同様の範囲
- ±1e+20 はそれぞれ -∞、+∞ を表す

ソルバー定義の接尾辞：診断のための2例

実行不可能性の診断：.iis

- cplex_options に iisfind 1(高速)/2(小さいIIS) を指定
- 既約実行不可能部分系(IIS)＝それ以上減らせない矛盾の集合
- .iis は記号接尾辞：non/low/fix/upp を返す

```
ampl: option cplex_options
ampl?    'iisfind 1';
ampl: solve;
Returning iis of 7 variables
  and 2 constraints.

ampl: print {i in 1.._ncons:
ampl?  _con[i].iis <> "non"}: _conname[i];
Diet['B2']    Diet['NA']
```

非有界方向：.unbdd

- 非有界LPで実行可能半直線 X + αd（α≥0）の方向 d を返す
- Benders分解の部分問題などで活用（trnloc1d.mod）
- 使用前に repeat ループの外で suffix unbdd OUT; の宣言が必要

```
ampl: solve;
CPLEX 8.0.0: unbounded problem.

ampl: display Supply_Price.unbdd;
Supply_Price.unbdd [*] :=
 1 -1  4 -1   7 -1 10 -1 ...
```

suffix 文の構文

```
suffix priority IN, integer,
   >= 0, <= 9999;
```

接尾辞の型

(指定なし)	任意の数値
integer	整数値
binary	0 または 1
symbolic	suffix_table に列挙された文字列

接尾辞は変数・目的・制約（および問題）の名前に付けられる独立した名前空間をもつ

入出力の指定

IN	AMPLが書き出し、ソルバーが読む
OUT	ソルバーが書き出し、AMPLが読む
INOUT	IN と OUT の両方（既定）
LOCAL	読み書きどちらもしない

記号接尾辞の仕組み

symbolic接尾辞ごとに数値版 _num が自動定義され、name_table オプションで整数と文字列の対応を宣言する

2つのモデルを交互に解く：切断ストック問題

Gilmore–Gomory の列生成手続き：切断最適化とパターン生成を交互に solve する

双対価格 *price[i]* → 新パターン *Use[i]*

切断最適化モデル (cut.mod)

```
minimize Number: Σ Cut[j]
subject to Fill{i}: Σ nbr[i,j]*Cut[j] >= orders[i]
```

新パターンを追加

パターン生成モデル (cut.mod)

```
minimize Reduced_Cost:
   1 - Σ price[i]*Use[i]
subject to Width_Limit: Σ i*Use[i] <= roll_width
```

```
需要を満たす初期パターンを選ぶ
repeat {
   (分数の)切断最適化問題を解く → price[i] := Fill[i].dual
   パターン生成問題を解く
   最適値 < 0 なら新しいパターンを追加、そうでなければ終了して整数解を求める
}
```

problem 文でモデルの一部を切り替える

- **problem 文が変数・目的・制約の部分集合に名前を付ける**
- 現在の問題になると、その部分集合以外の変数は現在値に固定、目的・制約は除去される
- fix/drop や 別モデルの solve を毎回書く方式より簡潔で誤りにくい
- 問題ごとに option relax_integrality 等の環境も自動的に切り替わる

```
# cut.run（図14-3・抜粋）
problem Cutting_Opt: Cut, Number, Fill;
   option relax_integrality 1;

problem Pattern_Gen: Use, Reduced_Cost,
   Width_Limit;
   option relax_integrality 0;

repeat {
   solve Cutting_Opt;
   let {i in WIDTHS} price[i] :=
      Fill[i].dual;
   solve Pattern_Gen;
   if Reduced_Cost < -0.00001 then {
      let nPAT := nPAT + 1;
      let {i in WIDTHS} nbr[i,nPAT]
         := Use[i];
   } else break;
}
```

名前付き問題の使用と表示

```
ampl: problem;
problem Initial;

ampl: problem Cutting_Opt:
ampl?    Cut, Number, Fill;
ampl: problem Pattern_Gen:
ampl?    Use, Reduced_Cost, Width_Limit;
ampl: problem;
problem Pattern_Gen;

ampl: show problems;
problems: Cutting_Opt  Pattern_Gen

ampl: show Cutting_Opt;
problem Cutting_Opt: Fill, Number, Cut;
```

- problem だけのコマンド：現在の問題名を表示（既定は Initial）
- show problems：定義済みの名前付き問題を一覧表示
- show <name>：問題を構成する変数・目的・制約を表示
- expand <name>：データ代入後の目的・制約を具体的に展開表示
- solve <name> は problem <name> に続けて solve するのと同じ効果
- redeclare problem／reset problem で定義を更新・巻き戻し可能

名前付き環境 (environ)

- **環境＝すべての AMPL オプション値の集まり**
- 既定では各 problem 文が同名の環境も自動的に作成する
- 環境は作成時点のオプション値を継承し、現在である間の変更を保持する
- environ 文で問題とは独立に環境を宣言・命名できる
- problem 文で "environ 名" を指定すれば既存の環境を結びつけられる
- 現在の問題／環境が切り替わるたびに option 設定も自動的に切り替わる

```
environ Master;

problem MasterII environ Master:
   Total_Cost, Weight, Multi;
```

cut.run の例では relax_integrality を問題ごとに切り替え、ループ本体を単純に保っている

まとめ

- 前処理は問題を自動的に簡約し、presolve/var_bounds/presolve_eps 等で挙動を制御できる
- solve_result・solve_result_num・solve_exitcode で求解結果や起動失敗を検査できる
- .sstatus/.astatus/.status で変数・制約の基底状態を調べ、再最適化を高速化できる
- suffix 文でユーザー定義・ソルバー定義の接尾辞を宣言し、高度な情報をやりとりできる
- problem 文と environ 文で、複数モデルを交互に解く手続きを簡潔かつ効率的に書ける

AMPL：数理計画のためのモデリング言語



第 15 章

ネットワーク線形計画問題

節点と枝でモデルを描き、*node* と *arc* で AMPL に語らせる

転送　最大フロー　最短路　割当　node / arc

AGENDA

本章の内容

15.1	**最小費用転送モデル**	節点・枝の集合、供給/需要、費用/容量、収支制約 Balance
15.2	**その他のネットワークモデル**	最大フロー・最短路・輸送/割当モデルへの応用
15.3	**node と arc によるネットワークモデルの宣言**	net_in / net_out、from・to・obj 句で図に沿って記述
15.4	**node と arc 宣言の規則**	収支条件の書式、側面制約・側面変数との組み合わせ
15.5	**ネットワーク線形計画問題を解く**	純粋ネットワーク構造と専用アルゴリズムによる高速化

ネットワークフローの基本構造

- 節点（node）＝円　… 都市・交差点などの地点
- 枝（arc）＝矢印　… ある節点から別の節点へ走る有向リンク
- フロー（flow）＝矢印の向きに沿って枝を移動する量
- 枝には「費用, 容量」の2つの数字が付く（図の例）
- **決定変数＝枝上のフロー量、制約＝上下限と各節点でのフロー保存の2種類だけ**

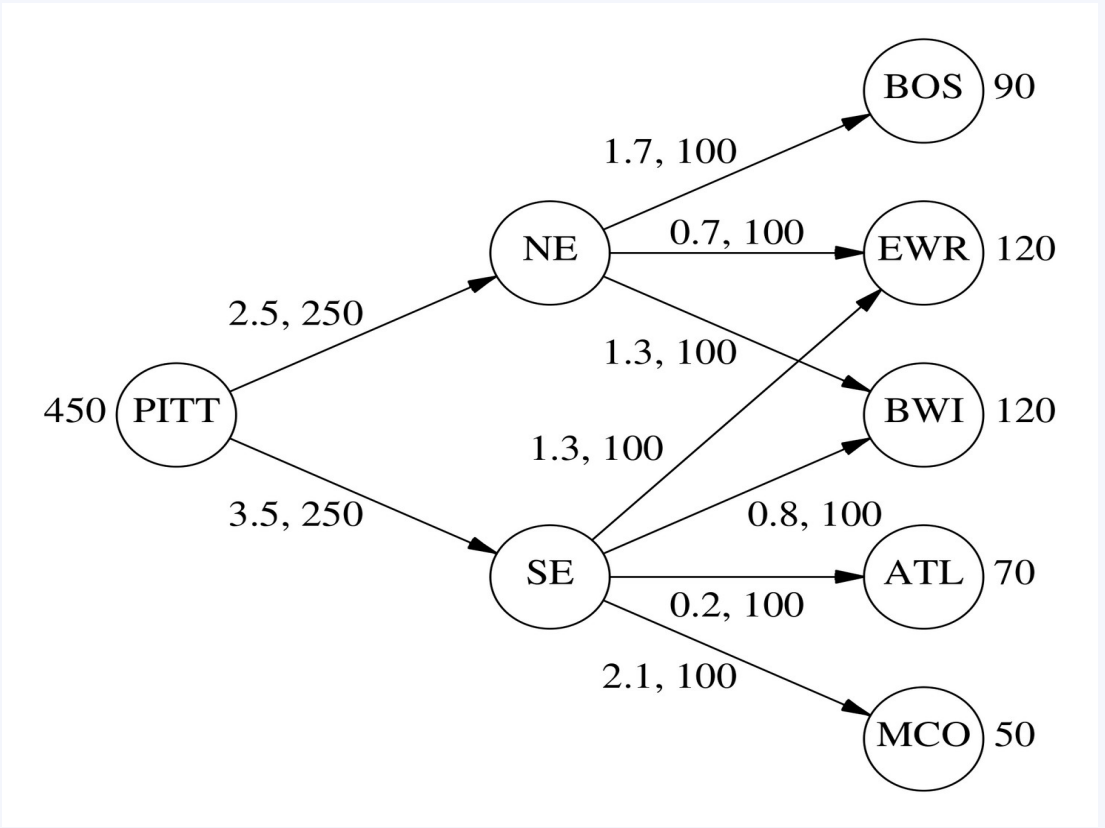


図15-1　有向ネットワーク（各枝の数字は「費用, 容量」）

例題：都市間パッケージ輸送

- PITT工場が来週450千パッケージを製造
- NE・SEの2配送センターが工場から受け取り、各倉庫へ転送
- BOS・EWR・BWI・ATL・MCOの5倉庫がそれぞれ必要量をもつ
- 各リンクには「1000パッケージ当たりの費用」と「輸送上限（容量）」がある
- **目的：容量を守り需要を満たす最小費用の輸送計画を求める**

倉庫ごとの需要（千パッケージ）

倉庫	需要
BOS	90
EWR	120
BWI	120
ATL	70
MCO	50

- 供給合計 450
- ＝
- 需要合計 450
- （90+120+120
- +70+50）

450
PITT の供給（千パッケージ）

5
倉庫（需要地点）の数

9
リンク（枝）の本数

一般的な転送モデル（1）— 宣言と目的関数

- 都市の集合 CITIES と、都市の順序対の部分集合 LINKS でリンクを表す
- 各都市に供給 supply と需要 demand（ともに ≥ 0）を定義
- 各リンクに費用 cost と容量 capacity を定義
- 変数 Ship[i,j] は非負、かつ容量以下に制限
- **目的：全リンクの cost × Ship の総和を最小化**

net1.mod（抜粋）

```
set CITIES;
set LINKS within (CITIES cross CITIES);

param supply {CITIES} >= 0;
param demand {CITIES} >= 0;

param cost {LINKS} >= 0;
param capacity {LINKS} >= 0;

var Ship {(i,j) in LINKS} >= 0,
   <= capacity[i,j];

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

一般的な転送モデル（2）— 収支制約 Balance

```
subject to Balance {k in CITIES}:
   supply[k]
    + sum {(i,k) in LINKS} Ship[i,k]
   = demand[k]
    + sum {(k,j) in LINKS} Ship[k,j];
```

- 各都市 k で「供給＋入るフロー」＝「需要＋出るフロー」
- sum{(i,k) in LINKS} Ship[i,k] … k に入るすべてのリンクの和
- sum{(k,j) in LINKS} Ship[k,j] … k から出るすべてのリンクの和
- **この添字づけの約束が収支制約を簡潔に書く鍵**

「正味需要」形式

```
# 変数を左辺、定数を右辺に移すと
subject to Balance {k in CITIES}:
     sum {(i,k) in LINKS} Ship[i,k]
   - sum {(k,j) in LINKS} Ship[k,j]
   = demand[k] - supply[k];
```

入る量 − 出る量 ＝ 正味需要

正の値：倉庫都市　／　負の値：工場都市　／　零：転送都市

特化した転送モデル

- 工場・配送センター・倉庫の3種類を別々の集合で宣言
- 工場は1つだけ → 記号パラメータ p_city で表す
- DW_LINKS は「配送センター→倉庫」の対だけに限定
- **不適切な配置（倉庫間リンクなど）を構造的に排除できる**
- 一般モデルの CITIES・LINKS・supply・demand はこれらから導出

net2.mod（抜粋）

```
param p_city symbolic;
set D_CITY;
set W_CITY;
set DW_LINKS within
   (D_CITY cross W_CITY);

param p_supply >= 0;
param w_demand {W_CITY} >= 0;

set CITIES =
   {p_city} union D_CITY union W_CITY;
set LINKS =
   ({p_city} cross D_CITY) union DW_LINKS;
```

転送モデルの変形

1 不等式の収支

供給が需要を上回りうるなら P_Bal の = を <= に。

```
subject to P_Bal:
  sum{i in D_CITY} PD_Ship[i] <=
p_supply;
```

2 輸送中の損失

損失率 pd_loss を導入し、届く量を (1-pd_loss)×Ship に。

```
param pd_loss {D_CITY} >= 0, < 1;
subject to D_Bal {i in D_CITY}:
  (1-pd_loss[i])*PD_Ship[i] =
    sum{(i,j) in DW_LINKS} DW_Ship[i,j];
```

3 単位の換算

需要が箱、輸送が千パッケージなら換算係数 ppc を使う。

```
param ppc integer > 0;
subject to W_Bal {j in W_CITY}:
  sum{(i,j) in DW_LINKS}
    (1000/ppc)*DW_Ship[i,j] =
w_demand[j];
```

最大フローモデル

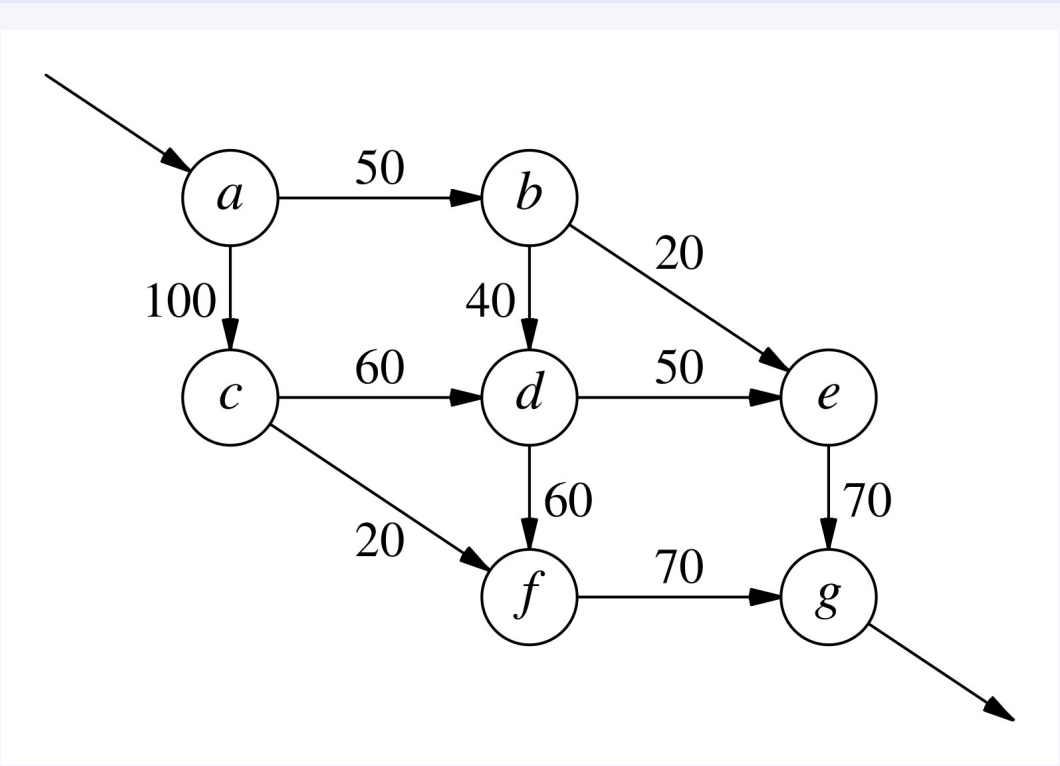


図15-5　交通流ネットワーク（数字は毎時の容量）

- a から入り g から出る最大交通流を求める

- 収支制約 Balance は入口・出口を除く各交差点だけに課す
- 目的：入口から出る総フロー Entering_Traff を最大化
- **最適値：毎時 130 台**

netmax.mod（抜粋）

```
set ROADS within
   (INTER diff {exit}) cross (INTER diff {entr});
param cap {ROADS} >= 0;
var Traff {(i,j) in ROADS} >= 0,
   <= cap[i,j];

maximize Entering_Traff:
   sum {(entr,j) in ROADS} Traff[entr,j];

subject to Balance {k in INTER diff {entr,exit}}:
  sum{(i,k) in ROADS} Traff[i,k]
   = sum{(k,j) in ROADS} Traff[k,j];
```

最短路モデル

- 同じネットワーク図（図15-5）を使い、容量を移動時間 time に読み替える
- 変数 Use[i,j] は「最短路に含まれるなら1、含まれなければ0」
- 目的：time × Use の総和（総移動時間）を最小化
- 追加の制約 Start：入口から流れ出る量をちょうど1に固定

netshort.mod（抜粋）

```
param time {ROADS} >= 0;
var Use {(i,j) in ROADS} >= 0;
minimize Total_Time:
   sum {(i,j) in ROADS} time[i,j]*Use[i,j];
subject to Start:
   sum {(entr,j) in ROADS} Use[entr,j] = 1;
```

AMPL 対話セッション

```
ampl: model netshort.mod;
ampl: solve;
MINOS 5.5: optimal solution found.
1 iterations, objective 140
ampl: option omit_zero_rows 1;
ampl: display Use;
Use :=
a b    1
b e    1
e g    1
;
```

最短路：a → b → e → g　（所要 140 分）

輸送モデルと割当モデル

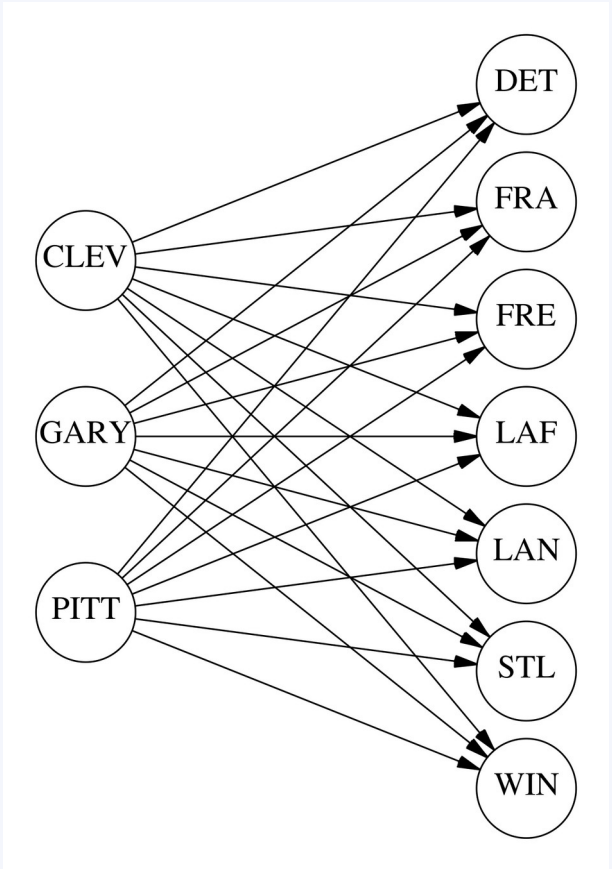


図15-8　二部ネットワーク

- 節点は発地群と着地群の2つに分かれる二部構造
- 枝はすべて発地→着地を結ぶ（発地間・着地間はなし）
- **このネットワーク上の最小費用転送＝輸送モデル／割当モデル**

netasgn.mod（抜粋）

```
set PEOPLE;  set PROJECTS;
set ABILITIES within
   (PEOPLE cross PROJECTS);

var Assign {(i,j) in ABILITIES}
   >= 0, <= limit[i,j];

subject to Supply {i in PEOPLE}:
  sum{(i,j) in ABILITIES}
     Assign[i,j] = supply[i];
```

- 係数は時間・人数・順位のいずれでもよい
- 順位を係数にすると「希望順位の合計最小化」割当になる

なぜ node と arc が必要か

- 代数的な var / subject to は変数と制約から出発し、節点・枝は式の中に暗黙に埋め込まれる
- ネットワークが複雑になるほど収支制約の式は読み取りにくくなる
- **node と arc は逆から出発：節点と枝を直接宣言し、フロー変数と収支制約を暗黙に生成**
- 図の見え方（節点が先、枝が後）にそのまま対応する

代数的な表現（読み取りにくい）

```
sum {(i,k) in LINKS} Ship[i,k]
  - sum {(k,j) in LINKS} Ship[k,j]
```

node / arc の表現（節点収支が一目瞭然）

```
net_in
```

「入るフロー − 出るフロー」を1語で表す組込みキーワード
（net_out はその符号反転）

node と arc で書き直す — 一般的な転送モデル

net1node.mod（抜粋）

```
# set / param 宣言は前と同じ

minimize Total_Cost;

node Balance {k in CITIES}:
   net_in = demand[k] - supply[k];

arc Ship {(i,j) in LINKS} >= 0,
   <= capacity[i,j],
   from Balance[i], to Balance[j],
   obj Total_Cost cost[i,j];
```

- **node Balance**
- 各 CITIES の要素に1つの節点。net_in = demand − supply で正味需要を指定
- **arc Ship**
- 上下限は var と同じ書式。from・to で結ぶ節点、obj で目的関数への係数を指定
- **minimize は名前だけでよい（項は arc 側の obj 句が供給）**

node 宣言 — 収支条件の書式

- 書式：node 名前 {添字式}: 収支条件;
- 収支条件は net_in（または net_out）と算術式を =, <=, >= で結ぶ
- 上下限つきの範囲条件（arith <= net <= arith）も書ける
- **収支条件を省略すると net_in = 0（＝ net_out = 0）が既定**

3種類の節点の例（net3node.mod）

```
node Plant: net_out = p_supply;
node Dist {i in D_CITY};
node Whse {j in W_CITY}:
   net_in = w_demand[j];
```

許される収支条件の形

```
net-expr = arith-expr
net-expr <= arith-expr
net-expr >= arith-expr
arith-expr = net-expr
arith-expr <= net-expr
arith-expr >= net-expr
arith <= net-expr <= arith
arith >= net-expr >= arith
```

net-expr は ± net_in（または net_out）に定数項 arith-expr を足したものに限られる。

arc 宣言 — from / to / obj 句

- 書式：arc 名前 {添字}, 上下限, from 節点, to 節点, obj 目的 係数;
- from・to はどちらか片方を省略してもよい（枝の片端がぶら下がる）
- 係数を from/to 句の末尾に付けると輸送損失・単位換算を表せる
- **if を使った添字式で条件つきに from / to を切り替えられる**

損失係数の例

```
arc PD_Ship {i in D_CITY} >= 0, <= pd_cap[i],
   from Plant, to Dist[i] 1-pd_loss[i],
   obj Total_Cost pd_cost[i];
```

条件つき from / to（最大フロー）

```
arc Traff {(i,j) in ROADS} >= 0,
   <= cap[i,j],
   from {if i = entr} Entr_Int,
   from {if i <> entr} Intersection[i],
   to   {if j = exit} Exit_Int,
   to   {if j <> exit} Intersection[j],
   obj Entering_Traff
      (if i = entr then 1);
```

15.4 node と arc 宣言の規則

付加制約と付加変数

付加制約 —— 多品種フロー

arc の名前は subject to の中で通常の変数として使える。製品ごとの枝流量に共同の容量制限をかけられる。

```
subject to Multi {(i,j) in LINKS}:
  sum {p in PRODS} Ship[p,i,j]
     <= cap_joint[i,j];
```

→ ネットワーク構造は壊れるが、費用・容量部分とは分離して書ける（*netmulti.mod*）。

付加変数 —— 原料からの生産

先に var で定義した通常変数を node の収支条件に含められる。

```
var Feed {f in FEEDS, k in CITIES}
   >= 0, <= limit[f,k];
node Balance {p in PRODS, k in CITIES}:
  net_out = supply[p,k] - demand[p,k]
   + sum{f in FEEDS} yield[p,f]*Feed[f,k];
```

→ *Feed[f,k] が製品ごとのネットワークを1つの線形計画問題に結びつける（netfeeds.mod）。*

純粋ネットワーク構造と解の速さ

- 各変数が高々2本の制約に、係数 +1 と −1 で1回ずつ現れる ＝「純粋な」ネットワークLP
- **供給・需要・上下限が整数なら、必ず全整数の最適解が存在**
- node / arc で書けば AMPL が自動でネットワーク構造をソルバーに伝達
- 専用のネットワークシンプレックス法で大幅に高速化されうる

代数的定式化（net3.mod）

```
CPLEX 8.0.0: optimal solution; objective 1819
1 dual simplex iterations (0 in phase I)
```

```
ampl: model net3node.mod; data net3.dat; solve;
CPLEX 8.0.0: optimal solution; objective 1819
Network extractor found 7 nodes and 7 arcs.
7 network simplex iterations.
```

同じ目的関数値 1819 だが…

node / arc 定式化ではネットワーク抽出器が7節点・7枝を検出し、専用のネットワークシンプレックス法が適用される。

まとめ

1	**ネットワーク線形計画問題**	決定変数＝枝のフロー、制約＝上下限とフロー保存だけの特別なLP
2	**転送・最大フロー・最短路・割当**	同じ変数/制約の型を、目的や収支条件の違いで使い分ける
3	**node と arc 宣言**	net_in / net_out と from・to・obj で図に沿って直接記述できる
4	**側面制約・側面変数**	多品種フローなど、ネットワークを超える拡張も自然に組み込める
5	**純粋ネットワーク構造**	整数最適解の保証と専用アルゴリズムによる高速な求解につながる

AMPL：数理計画のためのモデリング言語

第16章

列単位の定式化

行を追うか、列を追うか — 変数中心でモデルを組み立てる

投入産出モデル　|　スケジューリングモデル　|　obj / coeff 句の規則

AGENDA

本章の内容

16.1 投入産出モデル

行単位・列単位の両方で生産モデルを定式化し、obj句・coeff句を導入する

16.2 スケジューリングモデル

シフト勤務の割当問題を列単位で定式化し、集合被覆問題との関係を見る

16.3 列単位の定式化の規則

to_come・coeff句・obj句の一般規則と、多品種フローモデルへの応用

CONCEPT

行単位と列単位、二つの視点

AMPLは通常、`var` 宣言 → `maximize/minimize` → `subject to` の順に「行単位（制約指向）」で書く。しかし特定の線形計画問題では、一つの変数に関わるすべての係数をまとめて見るほうが単純になる。

行単位（制約指向）

- **var → maximize/minimize → subject to の順**
- **一つの制約について全変数の係数を並べる**
- **本書の大半の例で用いられる標準的な書き方**
- **特化した制約が多いモデルに向く**

列単位（変数指向）

- maximize/subject to → var の順
- 一つの変数について全制約の係数を並べる
- 第15章のnode/arc宣言もこの発想の一種
- obj句・coeff句をvar宣言に追加して実現

原料と活動：io[i,j] の意味

製鋼所や精製所のような複雑な操業では、ある生産単位の投入が他の単位の産出になる。原料集合 MAT と活動集合 ACT に対し、投入産出係数 io[i,j] を次のように定義する。



- io[i,j] > 0 ： 活動 j の1単位が生む原料 i の量（産出）
- io[i,j] < 0 ： 活動 j の1単位が消費する原料 i の量の負値（投入）
- io[i,j] = 0 ： 原料 i は活動 j に関与しない
- 変数 Run[j] は活動 j の運用水準（act_min[j] ≤ Run[j] ≤ act_max[j]）

まず行単位で定式化する

iorow.mod

```
set MAT;    # materials
set ACT;    # activities

param io {MAT,ACT};    # input-output coefficients
param revenue {ACT};
param act_min {ACT} >= 0;
param act_max {j in ACT} >= act_min[j];

var Run {j in ACT} >= act_min[j], <= act_max[j];

maximize Net_Profit:  sum {j in ACT} revenue[j] * Run[j];

subject to Balance {i in MAT}:
   sum {j in ACT} io[i,j] * Run[j] = 0;
```

読み方

- Balance[i] は原料 i の産出量と消費量が均衡することを表す
- 資源の購入・製品の販売も「活動」として表現し、Run[i]の下限・上限で量を制御
- revenue[j] は活動ごとの単位収入（購入・生産は負＝費用）

図16-1　行単位の投入産出モデル

obj句・coeff句で列単位に書き換える

iocol1.mod（抜粋）

```
maximize Net_Profit;

subject to Balance {i in MAT}: to_come = 0;

var Run {j in ACT} >= act_min[j], <= act_max[j],
   obj Net_Profit revenue[j],
   coeff {i in MAT} Balance[i] io[i,j];
```

図16-2　列単位の定式化

3つのポイント

- obj Net_Profit revenue[j] ： 目的 Net_Profit に項 revenue[j]*Run[j] を追加
- coeff {i in MAT} Balance[i] io[i,j] ： 各制約 Balance[i] に係数 io[i,j] を追加
- to_come は制約の「雛型」— 後続のvar宣言が係数を埋めていく

注意：obj/coeff句がNet_ProfitとBalanceを参照するため、`var` 宣言は `maximize/subject to` の後に置く必要がある。

2

販売活動を独立した変数に

完成原料 MATF を販売する変数 Sell[i] を新設。目的・制約の宣言は図16-2のまま変えず、変更はすべて var 宣言に反映される。

iocol2.mod（var宣言部）

```
var Run {j in ACT} >= act_min[j], <= act_max[j],
   obj Net_Profit -cost[j],
   coeff {i in MAT} Balance[i] io[i,j];

var Sell {i in MATF} >= sell_min[i], <= sell_max[i],
   obj Net_Profit revenue[i],
   coeff Balance[i] -1;
```

図16-3　販売活動を伴う列単位の定式化

利点

- **Sell[i] の投入は原料iのみ→coeff は Balance[i] -1 の1項だけでよい**
- **明示されない係数はすべて0とみなされる（零係数を書く必要がない）**
- **display Sell で販売量だけを容易に確認できる**
- **新しい活動（例：購入変数Buy）の追加も定義を1つ足すだけ**

シフトへの従業員割当

工場の生産はシフトに分かれる。各シフトに必要な人数が働くよう従業員を割り当てたいが、**5連勤は不可** など認められる週間スケジュールが限られる。そこで「従業員をスケジュールに割り当てる」問題として捉える。

```
set SHIFTS;              # shifts
param Nsched;            # number of schedules
set SCHEDS = 1..Nsched; # set of schedules
set SHIFT_LIST {SCHEDS} within SHIFTS;

param rate {SCHEDS} >= 0;      # 1人当たり賃金率
param required {SHIFTS} >= 0;  # シフトごとの必要人数
```

モデル化の要点

- **SHIFT_LIST[j] ： スケジュールjに含まれるシフトの集合**
- **変数 Work[j] ： スケジュールjに割り当てる人数**
- **各シフトiは、iを含むスケジュールに割り当てられた人数の合計でカバーされる**

行単位で書くと制約が扱いにくい

```
var Work {SCHEDS} >= 0;
minimize Total_Cost: sum {j in SCHEDS} rate[j] * Work[j];

subject to Shift_Needs {i in SHIFTS}:
   sum {j in SCHEDS: i in SHIFT_LIST[j]} Work[j]
      >= required[i];
```

左辺は「`i in SHIFT_LIST[j]` を満たすすべての j」にわたる和 — この`sum ... : 条件` という書き方の扱いにくさが、列単位への動機になる。

発想の転換

- データはスケジュール別（j）に整理されている
- 各スケジュールに変数も1つずつ対応する
- → 係数を「変数ごと」に指定する方が自然で、大規模例ではより効率的

coeff句 と cover句

sched.mod（図16-4）

```
minimize Total_Cost;
subject to Shift_Needs {i in SHIFTS}:
   to_come >= required[i];

var Work {j in SCHEDS} >= 0,
   obj Total_Cost rate[j],
   coeff {i in SHIFT_LIST[j]} Shift_Needs[i] 1;
```

図16-4　列単位のスケジューリングモデル

cover句（係数1を簡潔に指定）

```
var Work {j in SCHEDS} >= 0,
   obj Total_Cost rate[j],
   cover {i in SHIFT_LIST[j]} Shift_Needs[i];
```

係数の意味

- SHIFT_LIST[j]の各シフトiについて、Shift_Needs[i]で係数1をもつ
- cover句はfrom/toと同様、係数1を簡潔に指定するための専用句
- 航空会社の乗務員スケジューリングなど「集合被覆問題」の典型形

実例：126通りのスケジュール

月〜金1日3シフト・土2シフト、各日100/78/52人が必要という設定で `sched.mod` を解く。

AMPLセッション

```
ampl: model sched.mod; data sched.dat; solve;
MINOS 5.5: optimal solution found.
19 iterations, objective 265.6
ampl: display Work;
Work [*] :=
  10 28.8   30 14.4   71 35.6   106 23.2   123 35.6
  18 7.6    35 6.8    73 28     109 14.4
  24 6.8    66 35.6   87 14.4   113 14.4
;
```

265.6
最適目的値（延べ勤務量）

13 / 126
実際に使われたスケジュール数

271人
各解を切り上げた実行可能整数解

分数解を切り上げた271人が最良の整数解かどうかは、第20章の**整数計画**の技法で判定する。

2

to_come と coeff / obj 句の文法

雛型制約の3つの形

```
to_come
to_come + arith-expr
arith-expr + to_come
```

- const-expr 関係演算子 to_come の形が最も一般的（例：to_come >= required[i]）
- 雛型制約が変数を含む場合、それは行単位・列単位の混成モデルになる

var 宣言に追加できる句

- coeff {添字式} 制約名 arith-expr ： 制約係数を指定（複数可）
- obj 目的名 arith-expr ： 目的関数の係数を指定
- 添字式は {if logical-expr} の形も可（条件付きで係数を生成）
- arc宣言のfrom/to句に加えcoeff句も使用可（側面制約の列単位記述）

省略のルール：目的が後続の`var`宣言だけで指定される線形項の和なら、目的式は `to_come` のみとなり省略できる。

多品種フローモデルを完全に列単位で

第15章の多品種フローモデル（図15-13）を、arc 宣言の coeff 句で完全に列単位に書き直す。フロー収支は from/to 句、リンク総量の側面制約は coeff 句で表す。

netmcol.mod（図16-6）

```
minimize Total_Cost;

node Balance {k in CITIES, p in PRODS}:
   net_in = demand[k,p] - supply[k,p];

subject to Multi {(i,j) in LINKS}:
   to_come <= cap_joint[i,j];

arc Ship {(i,j) in LINKS, p in PRODS} >= 0, <= capacity[i,j,p],
   from Balance[i,p], to Balance[j,p],
   coeff Multi[i,j] 1.0,
   obj Total_Cost cost[i,j,p];
```

ポイント

- from/to でフロー収支制約の係数を、coeffで側面制約の係数を同時に指定
- Multi[i,j] はリンク合計流量の上限（cap_joint）を表す離型制約
- node/arc句とcoeff句は併用できる

図16-6　図15-13の列単位の定式化

✓ まとめ

1. 行単位（変数→目的→制約）に対し、列単位は「先に目的と制約、後から変数」の順で書く発想
2. var 宣言に obj句・coeff句を加えることで、変数ごとに目的係数・制約係数をまとめて指定できる
3. 制約・目的の雛型は to_come で示し、後続のvar宣言が係数を埋めていく
4. 投入産出モデルやスケジューリングモデルなど、1変数が複数制約に現れる問題（集合被覆問題を含む）と相性が良い
5. arc宣言のfrom/to/coeff句との併用で、行単位・列単位を柔軟に混成できる

AMPL：数理計画のためのモデリング言語



第17章

区分的線形計画問題

折れ点と傾きで費用構造を表す ―― *AMPL の <<breakpoints; slopes>>* 記法

本章の内容

17.1 **費用項**
固定数／可変数の部分区間をもつ区分的線形コストの表現

17.2 **よくある 2・3 部分区間の項**
柔らかい制約・実行不可能性・可逆的活動のモデル化

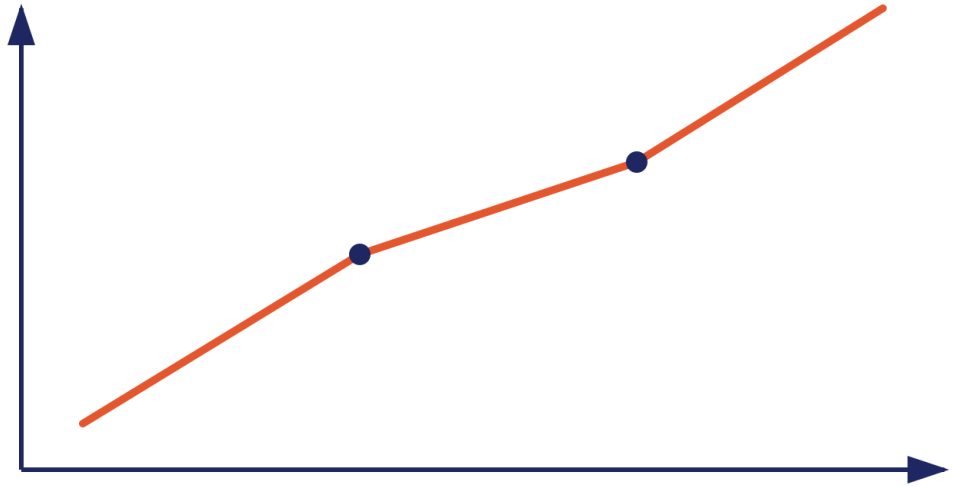
17.3 **他の区分的線形関数**
if-then-else、max、abs との等価性と記法の使い分け

17.4 **区分的線形最適化のための指針**
凸性・凹性と扱いやすさ、pl_linearize オプション

区分的線形とは

つながった線形の部分区間を継ぎ合わせてできる関数

- 線形計画問題の中には、厳密には線形でないが、直線の部分区間をつなぎ合わせてできる「区分的線形」の項を使うものがある。
- この形は思い描くのは容易だが、通常の代数記法で書くのは難しい。
- そこで AMPL は区分的線形項を記述する専用の簡潔な構文を提供する。
- 本章では、費用の記述・柔らかい制約・実行不可能性の分析・可逆的活動のモデル化という代表的な用途を、輸送モデルと生産モデルを例に見ていく。



図：傾きが変わる点（折れ点）をつないだ区分的線形関数のイメージ

固定数の部分区間 ——輸送コストの例

- 線形輸送モデルでは同じ単位費用が輸送量によらず一定。
- 現実にはより有利な料金は限られた単位数までしか適用されず、超過分にはより高い料金が課される。
- 各発地・着地の対に 3 つの費用水準（rate1, rate2, rate3）と 2 つの上限（limit1, limit2）を指定する。
- その結果、総輸送費用は輸送量とともに 3 つの部分区間をもつ区分的線形のやり方で増加する（図 17-1）。

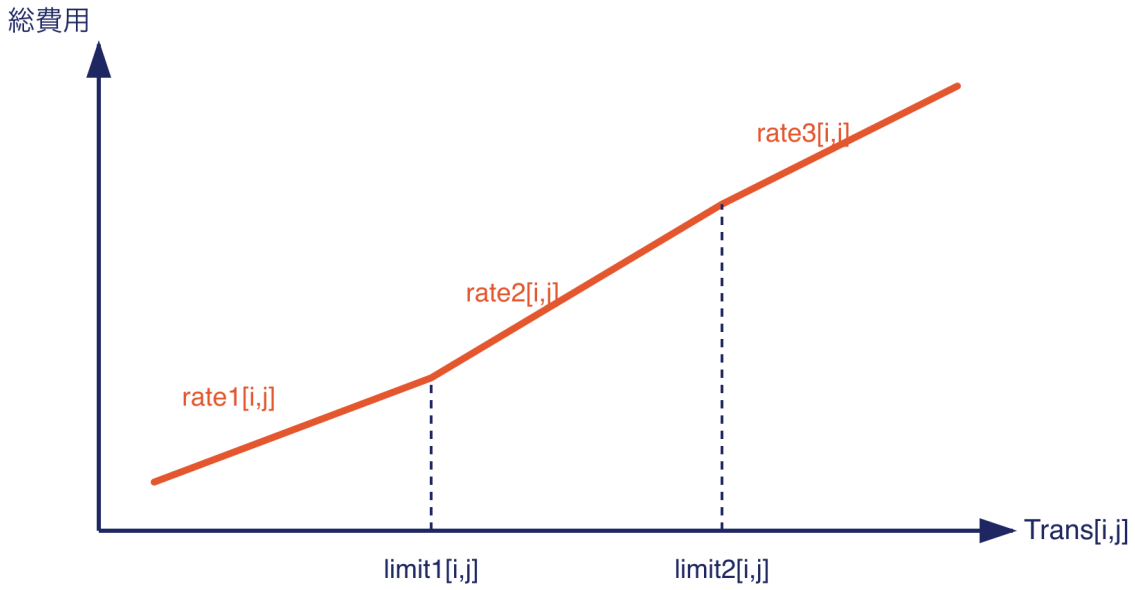


図17-1　3 つの傾きをもつ区分的線形関数

AMPL の区分的線形記法

```
<<limit1,limit2 ;   rate1,rate2,rate3 >>Trans
```

折れ点リスト　　傾きリスト

- « と » の間の式が区分的線形関数を記述し、続けて適用する変数名を書く。
- 折れ点リストと傾きリストはセミコロン（;）で区切り、各リストの要素はコンマ（,）で区切る。
- 傾きの数は折れ点の数よりちょうど 1 つ多くなければならない（最初の傾きは最初の折れ点より前、最後の傾きは最後の折れ点より後に適用）。
- 曖昧さを避けるため、AMPL は区分的線形関数が「零で零」の値をとると仮定する（既定の基準点）。

モデル例：transpl1.mod（3 部分区間）

```
set ORIG;   set DEST;
param supply {ORIG} >= 0;   param demand {DEST} >= 0;

param rate1 {i in ORIG, j in DEST} >= 0;
param rate2 {i in ORIG, j in DEST} >= rate1[i,j];
param rate3 {i in ORIG, j in DEST} >= rate2[i,j];
param limit1 {i in ORIG, j in DEST} > 0;
param limit2 {i in ORIG, j in DEST} > limit1[i,j];

var Trans {ORIG,DEST} >= 0;

minimize Total_Cost:
   sum {i in ORIG, j in DEST}
      <<limit1[i,j], limit2[i,j];
        rate1[i,j], rate2[i,j], rate3[i,j]>> Trans[i,j];
```

可変数の部分区間 —npiece によるデータ駆動

- 部分区間が 12 個などになると、rate1〜rate12 を個別に宣言するのは扱いにくい。
- 部分区間数そのものをパラメータ npiece[i,j] にするのがよい手法。
- 料金・上限をリンクと部分区間の組合せで添字づける：
- rate[i,j,p]、limit[i,j,p]。
- 折れ点・傾きのリストに添字式を使い、部分区間数をデータで変えられるようにする。

```
param npiece {ORIG,DEST} integer >= 1;

param rate {i in ORIG, j in DEST,
            p in 1..npiece[i,j]} >= ...;
param limit {i in ORIG, j in DEST,
             p in 1..npiece[i,j]-1} > ...;

minimize Total_Cost:
   sum {i in ORIG, j in DEST}
    <<{p in 1..npiece[i,j]-1} limit[i,j,p];
      {p in 1..npiece[i,j]} rate[i,j,p]>>
      Trans[i,j];
```

「柔らかい」制約への罰則項

第 4 章の複数週生産モデルを拡張

- 硬い制約「使用時間 ≤ avail[t]」を、望ましい水準を超えると罰則がかかる柔らかい制約に置き換える。
- avail_min[t]：罰則なしで使える時間
- avail_max[t]：使える上限時間
- time_penalty[t]：avail_min を超えた 1 時間ごとの利益減少
- 新変数 Use[t] （0 ≤ Use[t] ≤ avail_max[t]）を導入し、Time 制約を等式に変更する。

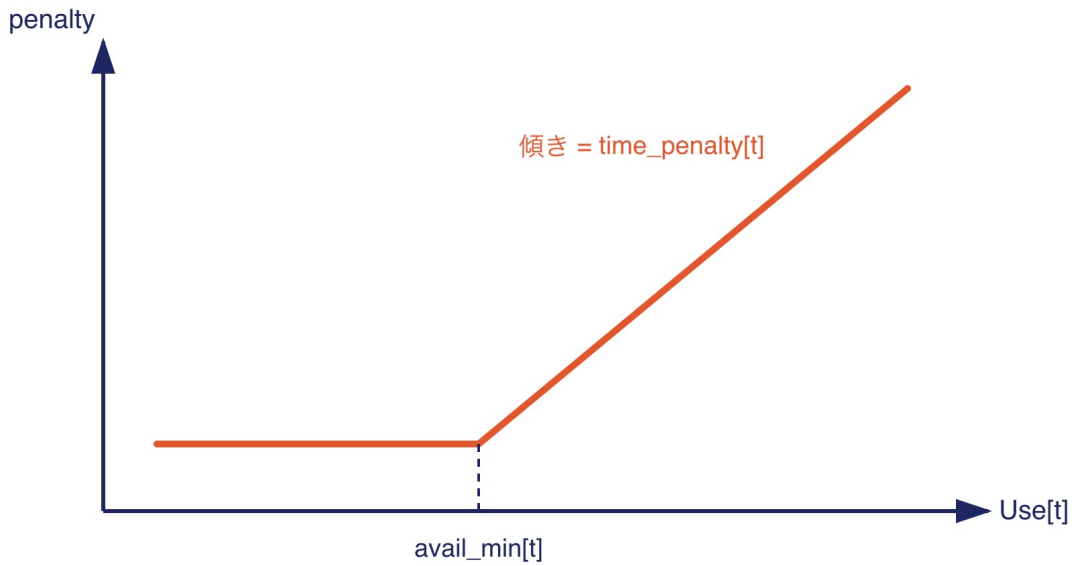


図17-4　利用時間に対する区分的線形の罰則関数（1 折れ点）

罰則項を組み込んだ目的関数と解

```
maximize Net_Profit:
   sum {p in PROD, t in 1..T}
      (revenue[p,t]*Sell[p,t]
       - prodcost[p]*Make[p,t]
       - invcost[p]*Inv[p,t])
  - sum {t in 1..T}
      <<avail_min[t]; 0,time_penalty[t]>>
      Use[t];
```

AMPL 対話セッション

```
ampl: model steelpl1.mod; data steelpl1.dat;
ampl: solve;
MINOS 5.5: optimal solution found.

ampl: display avail_min,Use,avail_max;
: avail_min Use avail_max :=
1     35      35     42
2     35      42     42
3     30      30     40
4     35      42     42 ;
```

第 1・3 週は罰則のない時間だけを使うが、第 2・4 週では罰則のある時間も使う。区分的線形計画問題の解では、多くの変数がいずれかの折れ点に「くっつく」性質がしばしば見られる。

実行不可能性への対処 ①：時間不足として見る

- 確約量（commit）を引き上げると時間が足りず、ソルバーが infeasible を返すことがある。
- 返る「実行不可能な解」は在庫が負になったり最小量を割り込んだりと、あちこちに散らばり原因がつかみにくい。
- 対策：avail_max[t] を折れ点にし、その先に非常に大きな傾き（例：100,000）を追加する。
- これにより Use[t] は avail_max を超えられるが、超過には極めて重い罰則がかかるため、必要最小限しか超えない。

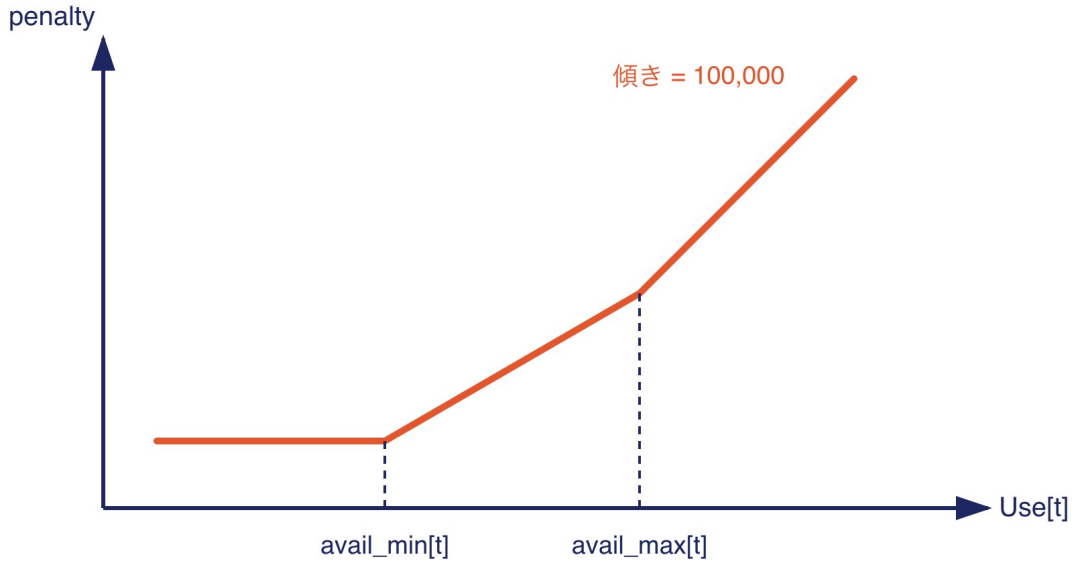


図17-6　2 つの折れ点をもつ利用時間の罰則関数

実行不可能性への対処 ②：確約超過として見る

- Sell[p,t] が commit[p,t] を下回る単位ごとに大きな罰則を引く（図 17-7）。
- 折れ点は零で零になるという既定の仮定があるため、単純な <<commit[p,t]; -100000,0>> だけでは値がずれる。

```
<<commit[p,t]; -100000,0>>
   (Sell[p,t], commit[p,t]);

# 第2引数 zero-expr で
# 「関数が commit[p,t] で零」と
# 明示できる
```

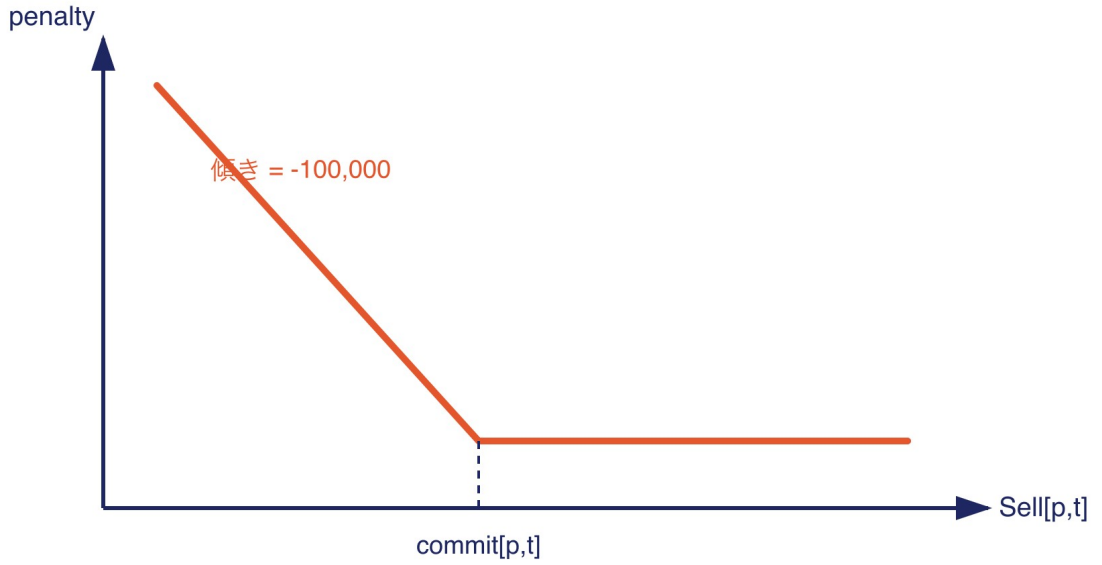


図17-7　販売量に対する罰則関数（commit[p,t] に折れ点）

⇄ 可逆的な活動 ——在庫と受注残

- Inv[p,t] の宣言から >= 0 を外すと、負の値を「受注残」として解釈できる。
- しかし単純な線形費用 invcost[p]*Inv[p,t] のままでは、負の Inv が利益を見かけ上増やしてしまい、不自然な解になる。
- 適切な費用関数は零を折れ点とし、正負どちらの方向に離れても増加する（図 17-8）。
- 傾きは -backcost[p]（負側）と invcost[p]（正側）：<<0; -backcost[p], invcost[p]>> Inv[p,t]。

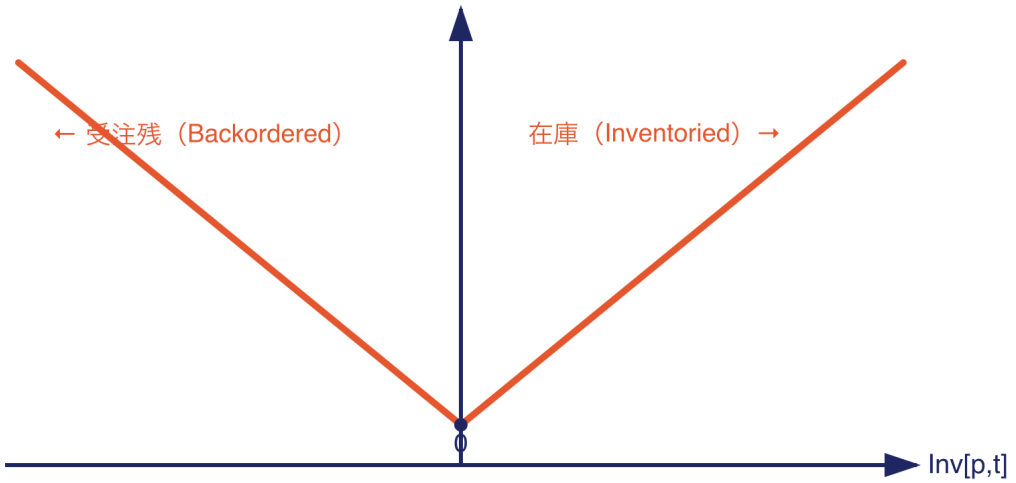


図17-8　在庫費用関数（零に折れ点をもつ V 字形）

他の書き方との等価性 ——if・max・abs

同じ関数でも解けるかどうかは記法次第

```
# 同じ関数の別表現
if Use[t] > avail_min[t]
then time_penalty[t]*
     (Use[t]-avail_min[t])
else 0

max(0, time_penalty[t]*
  (Use[t]-avail_min[t]))
```

AMPL はこれらの式が区分的線形だと自動検出しない –– 満足のいく解は得にくい

```
# 明示的な区分的線形記法
<<avail_min[t]; 0,time_penalty[t]>>
  Use[t]

# abs も同様
time_penalty[t] *
  <<avail_min[t]; -1,1>> Use[t]
```

構造がソルバーに直接伝わり、*LP として効率よく解ける*

分離不可能な例 ——最小最大の割当

- 目的 max{i} sum{j} cost[i,j]*Assign[i,j] は区分的線形だが、個々の変数について分離可能ではないため <<…>> 記法では書けない。
- 解決策：最大値を表す新変数 M を導入し、線形計画問題に書き直す。
- M はすべての i について右辺以上という制約 M_def を課し、目的は minimize M とする。
- 同じ考え方は「最大最小」目的にも適用でき、その場合は maximize と M <= … を使う。

```
var M;
minimize Max_Cost: M;

subject to M_def {i in PEOPLE}:
   M >= sum {j in PROJECTS}
         cost[i,j]*Assign[i,j];
```

区分的線形最適化のための指針

```
<<breakpoint-list; slope-list>> pl-argument
```

凸関数（傾きが非減少）

分離可能な凸区分的線形関数の最小化は LP として効率よく解ける（本章の例はすべてこの型）。

凹関数（傾きが非増加）

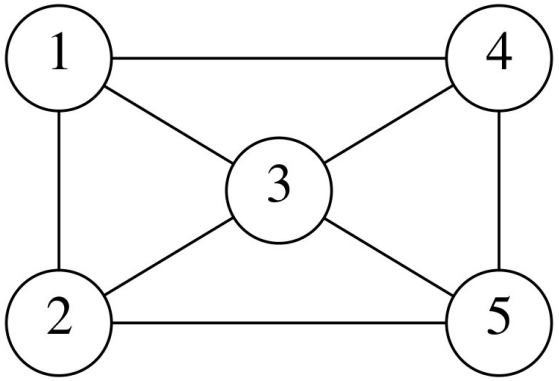
分離可能な凹区分的線形関数の最大化も同様に扱いやすい。

- この 2 つを外れる場合、分離可能な区分的線形の最適化は整数計画（第20章）の応用として扱われ、AMPL が等価な整数計画に自動変換する。
- 制約中の区分的線形は「凸で ≤ 制約左辺」または「凹で ≥ 制約左辺」の場合にかぎり LP で扱える。
- 直接扱えるソルバーがあれば option pl_linearize 0; で変換を止められる。
- 部分区間の総数が増えるほど難しさは増す ―― 10 数個を超えるなら第18章の非線形関数も検討する。

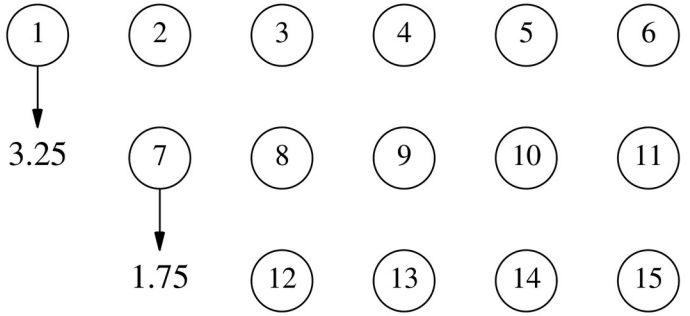
応用：構造設計への拡張（練習問題 17-7）

abs() の区分的線形記法は構造の最小重量設計にも使える

- 継手（joint）を棒（bar）でつないだ平面構造の最小重量設計問題。
- 棒の応力 F(i,j) は張力（正）・圧縮（負）のどちらもあり得る。
- 重量は w(i,j) = l(i,j) × |F(i,j)| ―― 絶対値が登場する。
- 均衡制約のもとで総重量を最小化する線形計画問題として、abs 項を区分的線形記法 <<0; -1,1>> で表現できる。



継手1〜5と棒の例



15 継手のトラス構造と外力の例

区分的線形記法は輸送や生産計画だけでなく、絶対値・逸脱量が現れるあらゆる線形計画（構造設計、回帰、割当など）に応用できる。

まとめ

第17章　区分的線形計画問題

1. **<<breakpoint-list; slope-list>> pl-argument**

   折れ点と傾きをセミコロンで区切って書く AMPL 独自の構文で、区分的線形の費用・罰則を簡潔に表現できる。

2. **データ駆動の部分区間数**

   npiece のような添字パラメータと {p in 1..npiece[i,j]} の添字式で、部分区間数を可変にできる。

3. **柔らかい制約・実行不可能性・可逆的活動**

   少数の折れ点をもつ罰則関数として、超過ペナルティや在庫／受注残の費用を自然にモデル化できる。

4. **if/max/abs より <<...>> を**

   同じ式でも明示的な区分的線形記法にしないと、AMPL は構造を検出できず効率よく解けない。

5. **扱いやすさは凸性・凹性で決まる**

   凸最小化／凹最大化は LP で自動的に解ける。それ以外は整数計画に変換され、部分区間が増えるほど難しくなる。

次章：第18章 非線形計画問題

AMPL：数理計画のためのモデリング言語



第18章　非線形計画問題

滑らかな非線形モデルの記述と、局所最適解に潜む落とし穴

AGENDA

本章の内容

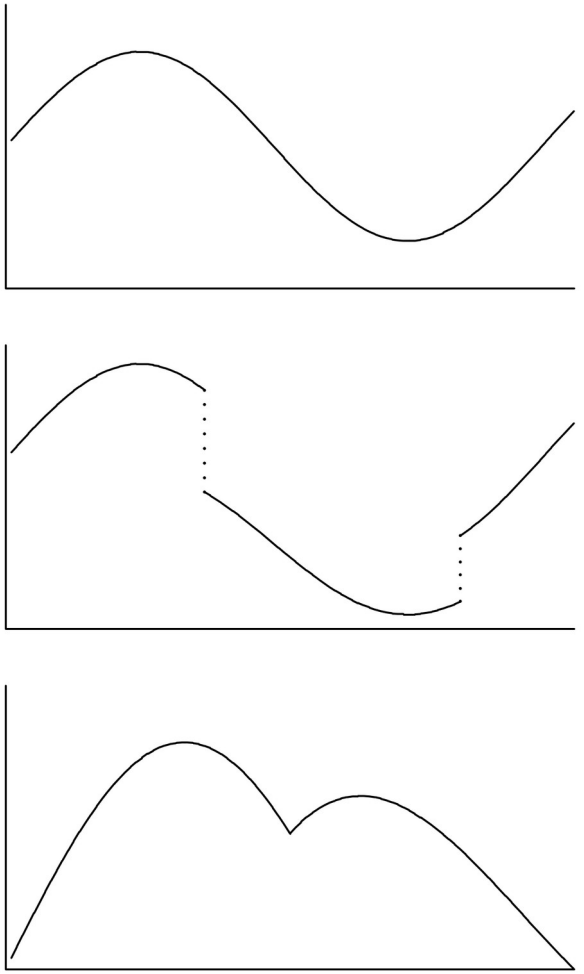
18.1	非線形性の発生源	仮定の緩和・望みの効果・物理過程という 3 つの経路
18.2	非線形変数	初期値の指定と、変数を自動的に代入で除去する機能
18.3	非線形の式	滑らかさを保つ書き方と AMPL 組み込み関数・外部関数
18.4	非線形計画法の落とし穴	値域違反、複数の局所最適解、その他の失敗の様式

非線形計画問題とは何か

- 連続変数（整数・離散ではない）を扱う点は線形計画問題と同じ
- 目的関数・制約の式は線形である必要はないが、「滑らか」でなければならない
- 滑らか＝すべての点で連続かつ勾配（1階微分）が一意に定まること
- **跳躍（不連続）や折れ目（微分不可能）をもつ関数は対象外**

なぜ特別扱いされるのか

他の「線形でない」問題と区別しやすく、応用が広く、確立されたアルゴリズムで解けるため。ただし同程度の線形計画問題より定式化・求解ははるかに難しい。



(a) 滑らかで連続　(b) 不連続　(c) 微分不可能



非線形性が持ち込まれる 3 つの経路

1



線形性の仮定を外す

輸送費用が数量に比例するといった仮定を緩め、区分的線形や滑らかな非線形の費用関数に置き換える

2



望みの効果を構成する

混雑による所要時間の急増など、線形関数では表せない振る舞いを狙って非線形関数を設計する

3



本質的に非線形な過程

パイプラインの圧力と流量の関係など、物理法則そのものが非線形な現象をそのまま定式化する



出発点：線形ネットワーク流モデル

- 都市 i から j への輸送費用は cost[i,j]、輸送量は Ship[i,j]
- LINKS は輸送経路が存在する都市の対の集合
- 各都市でフローが均衡するという制約をもつ
- この線形モデルを土台に、非線形の変種を組み立てていく

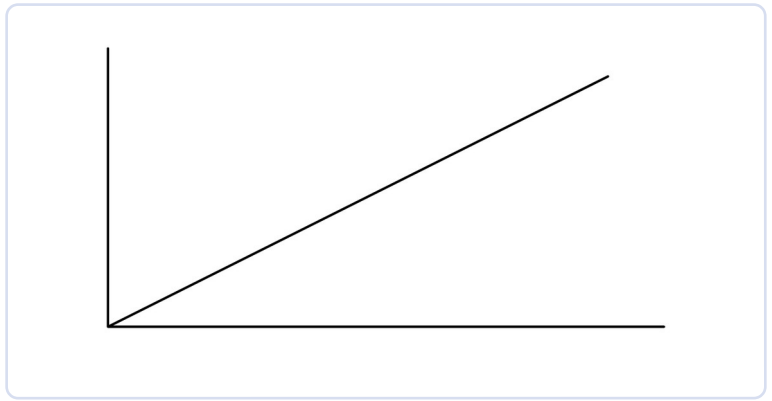
AMPL — net1.mod

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

subject to Balance {k in CITIES}:
   supply[k] + sum {(i,k) in LINKS} Ship[i,k]
      = demand[k] + sum {(k,j) in LINKS} Ship[k,j];
```

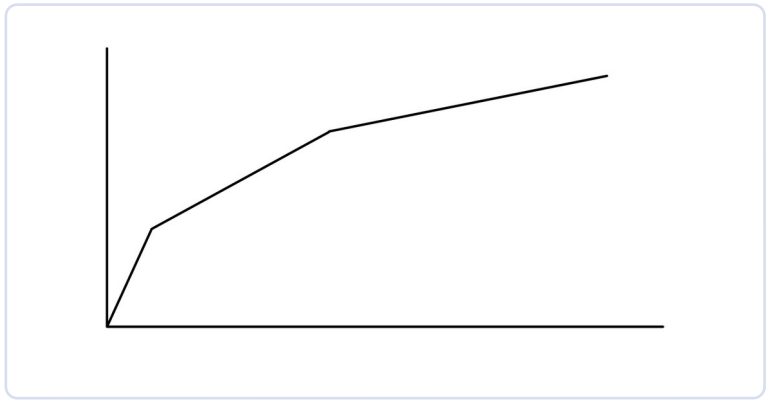
線形性の仮定

i から j への輸送は、どの単位も同じ単価 cost[i,j] を要すると仮定している。
この仮定を緩めることが、非線形化の第一歩になる。

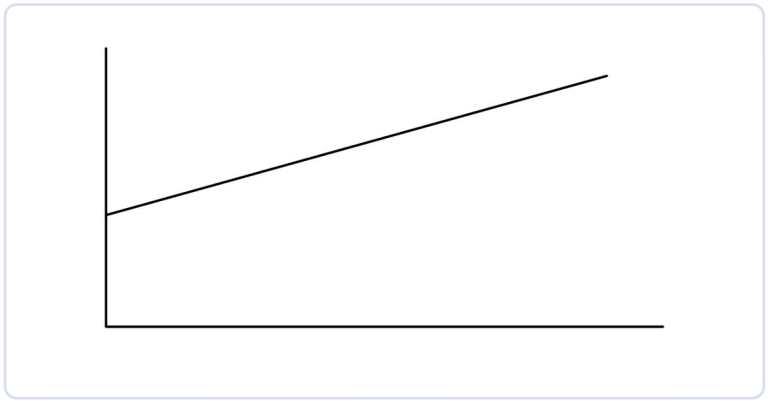
費用関数の形：線形から非線形へ



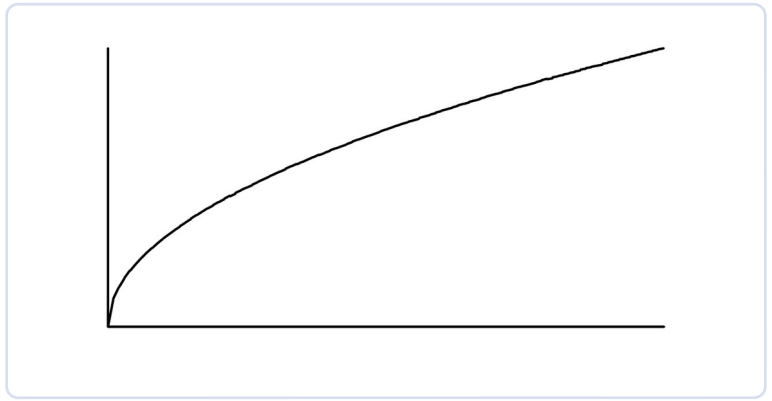
(a) 線形：一定の単価



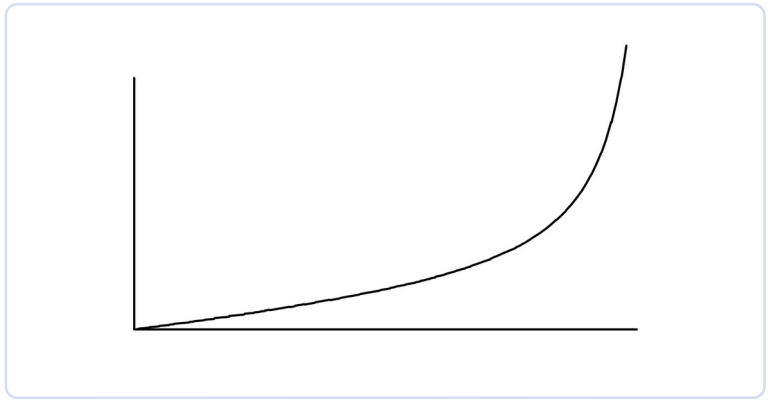
(b) 区分的線形：単価が段階変化



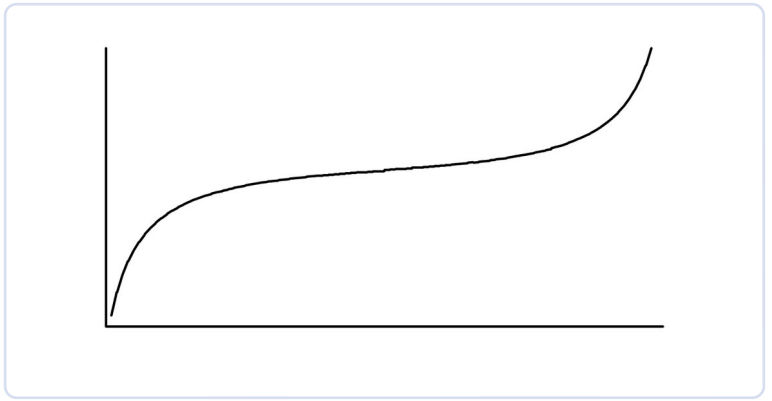
(c) 固定費用：跳躍あり



(d) 凹：数量割引に類似



(e) 凸：上限に近づき急増



(f) 凹凸混在

非線形費用関数の定式化

- cost[i,j]*Ship[i,j] を非線形の式に置き換えるだけでもよい
- 費用を変数 Cost[i,j] にすると、傾きが数量とともに連続変化する状況を表せる
- Ship がゼロ近くでは cost1 に近づき、大きくなると cost2 に落ち着く
- cost1 > cost2 なら図18-2d のような凹の費用曲線が得られる

変数の積 → 非線形

```
var Cost {ORIG,DEST};    # 単価
var Ship {ORIG,DEST} >= 0; # 輸送量

minimize Total_Cost:
   sum {i in ORIG, j in DEST} Cost[i,j] * Ship[i,j];
```

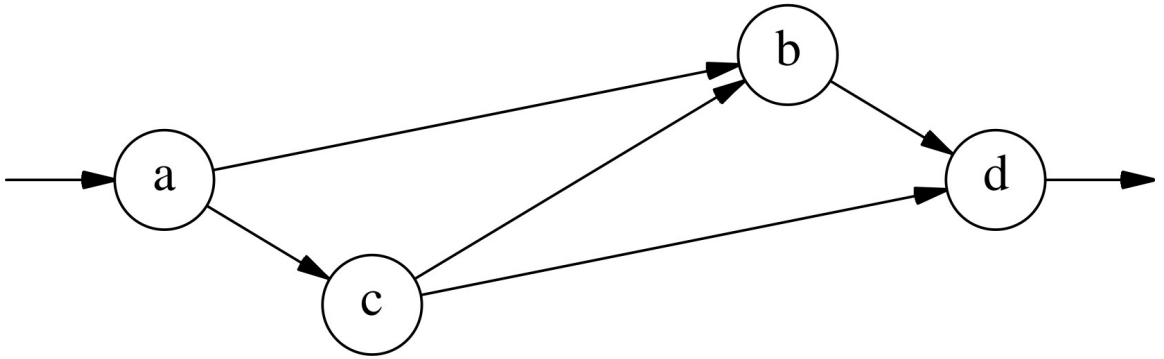
除算を含む方程式

```
subject to Cost_Relation {i in ORIG, j in DEST}:
   Cost[i,j] =
     (cost1[i,j] + cost2[i,j]*Ship[i,j])
     / (1 + Ship[i,j]);
```

ポイント：
積 Cost[i,j]*Ship[i,j] や比の式は AMPL の線形性の規則を満たさないため「非線形」として扱われる。式そのものは自由に書けるが、意味のある滑らかな関数になっているかは利用者の責任。

望みの効果を得る：交通混雑モデル

- 所要時間は交通量が少ないうちはほぼ一定
- リンクの容量 cap[i,j] に近づくにつれ無限大へ急増させたい
- この振る舞いは線形関数では表現できない
- sens[i,j] が両極端の間の形状を左右する



交差点 a→b→d、a→c→d を経由する交通流ネットワーク（演習18-4 で詳しく扱う）

所要時間関数

```
time[i,j] +
  (sens[i,j]*Ship[i,j])
  / (1 - Ship[i,j]/cap[i,j])
```

常に凸な関数

Ship[i,j] が小さい値のときは time[i,j] にほぼ等しく、cap[i,j] に近づくにつれて無限大に向かう。図18-2e に似た凸のグラフをもつ。



望みの効果を得る：需要乖離への罰則

- 変数 Discrepancy[k] で実配送量と需要量の乖離を表す（正負どちらも可）
- 乖離が大きくなり過ぎないよう目的関数に罰則項を追加する

区分的線形の罰則

```
minimize Total_Cost:
  sum{(i,j) in LINKS} cost[i,j]*Ship[i,j] +
  sum{k in CITIES}
     pen * <<-1,1; 0>> Discrepancy[k];
```

2次の罰則

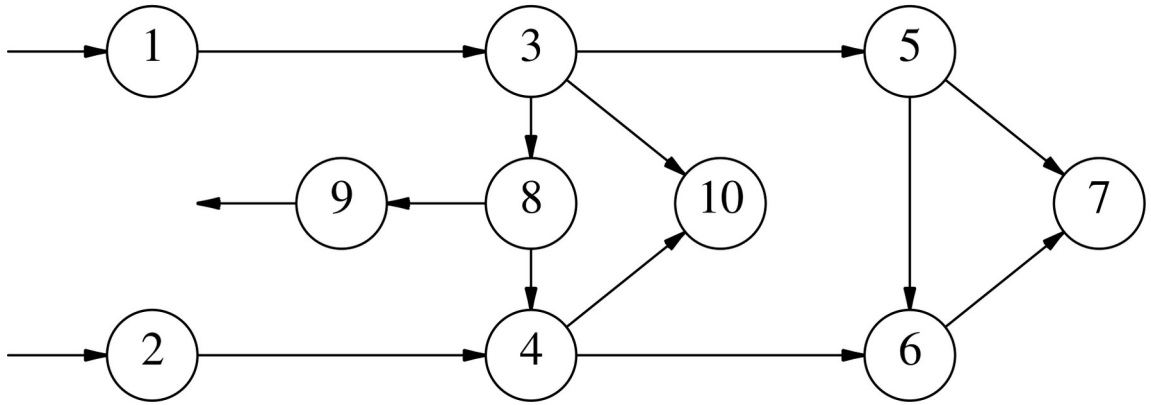
```
minimize Total_Cost:
  sum{(i,j) in LINKS} cost[i,j]*Ship[i,j] +
  sum{k in CITIES}
     pen * Discrepancy[k] ^ 2;
```

乖離が比例的に罰せられるだけで、単位ごとの罰則は一定。大きな乖離を十分には抑止できない。AMPL はこの目的関数を難なく線形に変換する。

乖離が悪化するほど単位当たりの罰則が着実に増大する。近似・データ当てはめでよく使われる「2乗和」の典型例。

本質的に非線形な過程：天然ガスパイプライン

- 流量 Ship[i,j] に加え、各都市 k の圧力 Press[k] も変数として定義
- 圧力が高い都市から低い都市へ流れが向かう
- c[i,j] はパイプの長さ・直径・効率・ガスの性質で決まる定数
- 圧縮機や弁はさらに別の非線形の流量関係を生む



流量・圧力の関係

```
Flow[i,j]^2 =
   c[i,j]^2 * (Press[i]^2 - Press[j]^2)
```

物理モデルの非線形性は経済ではなく物理法則・実測から生じるため形は特定しやすい一方、より複雑な関数形や変数間の相互作用を伴いやすい。

非線形変数：初期値の重要性

- solve すると初期値はソルバーへの出発点の推測として渡される
- 初期値を与えなければ AMPL は暫定的にゼロを設定する
- 良い初期値はソルバーの速度だけでなく、見つかる解そのものを左右し得る

宣言時に指定

```
var Ship {LINKS} >= 0, := 1;

var Ship {(i,j) in LINKS} >= 0,
   := cap[i,j] - 1;
```

data 文で指定

```
var Ship:   FRA   DET   LAN   WIN :=
  GARY      800   400   400   200
  CLEV      800   800   800   600
  PITT      800   800   800   200 ;
```

双対変数にも同様の機能

制約名への代入は、対応する双対変数（ラグランジュ乗数）の初期値として解釈される。MINOS などのソルバーはこれらを利用できる。

非線形変数：自動代入による変数の除去

- 定義変数（= の左辺に単一の変数をもつ制約）は代入で除去できる
- 除去しても変数名は結果の閲覧時に引き続き利用できる
- 式を複雑にする代わりに変数・制約の数を減らせる — 効果はモデル次第

方法1：オプション

```
option substout 1;
solve;

#  「定義」制約の条件：
#  ・左辺に変数がちょうど1つ
#  ・その変数に上下限等の制限なし
#  ・既に代入済みの制約に現れない
```

方法2：= 句で明示

```
var Cost {(i,j) in LINKS}
   = (cost1[i,j] +
      cost2[i,j]*Ship[i,j])
     / (1 + Ship[i,j]);
```

非線形部分は1つだけ保持し代入先リストを添える（オプション linelim=0 で無効化可）ため、線形項のみが実際に複製される。



非線形の式：滑らかさを壊す落とし穴

- AMPL の式構文は自由度が高いため、書けてしまうことと滑らかであることは別問題
- abs・min・max を変数に適用すると一般に滑らかでなくなる
- if_then_else も、変数を含む条件を伴うと不連続になりやすい

滑らかでない例 ①

```
cost[i,j] * abs(Flow[i,j])
```

ゼロで傾きが急変（折れ目）

滑らかでない例 ②

```
if Flow[i,j] = 0 then 0
else base[i,j]+cost[i,j]*Flow[i,j]
```

ゼロで値が跳躍（不連続）

AMPL は文句を言わない

こうした式を使ってもエラーにはならず、ソルバーが「最適解」を返すことさえある。だが結果は誤っている可能性がある。滑らかさの保証はモデラーの責任。

非線形の式：滑らかさを保つ工夫

- 流量の符号で向きを表せば、1対の方程式を1つの if 式にまとめられる
- Flow=0 で2つの2次関数が滑らかに「出会う」ため結果は滑らか

対称的な if の活用

```
(if Flow[i,j] >= 0 then
    Flow[i,j]^2
 else -Flow[i,j]^2)
  = c[i,j]^2*(Press[i]^2-Press[j]^2)
```

特異点の回避

```
if abs(Flow[i,j]-1) > 0.00001 then
   log(Flow[i,j])/(Flow[i,j]-1)
else
   1.5 - Flow[i,j]/2
```

Flow[i,j]の符号で i→j／j→i の向きを判定。定数 c[i,j]=c[j,i] を利用し1本の制約に統合できる。

log(x)/(x-1) は x=1 で 0/0 になりエラー。厳密には滑らかでなくても、数値的には十分滑らかな線形近似に置き換える。

落とし穴①：関数の値域違反

- rate*Trans/(1-Trans/limit) を最小化する非線形輸送モデルで発生
- ソルバーが Trans[i,j] = limit[i,j] を許し、ゼロ除算でエラー
- 出発点を変えても Trans が limit を超え、目的関数が -10^{18} 規模に発散

amplsession

```
ampl: solve;
MINOS 5.5 Error evaluating objective Total_Cost
can't compute 8000/0
MINOS 5.5: solution aborted.
```

修正：上限を課す

```
var Trans {i in ORIG, j in DEST}
   >= 0, <= .9999 * limit[i,j];

# 修正後 → optimal solution found
```

教訓

非線形関数は変数の意図した範囲の外では非常に悪い振る舞いをし得る。特異点（limit[i,j]）から十分に遠ざける上限（.9999倍）を明示的に与える必要がある。



落とし穴②：複数の局所最適解

- Trans^0.8 を導入すると凹凸混在の費用曲線になる
- 出発点 alpha=0.5 → 目的値 355,438　／　alpha=0 → 目的値 427,568
- どちらも局所最適だが、大域最適とは限らない
- 目的関数の凸性＋制約の線形性が揃えば局所最適＝大域最適が保証される

355,438
alpha = 0.5 の最良解

427,568
alpha = 0 の局所解

alpha を変えたときの Total_Cost（一部抜粋）

α=0.0	α=0.2	α=0.5	α=0.6	α=0.9	α=1.0
427568.1	366791.2	355438.2	356531.5	402795.3	365827.2

複数の出発点を系統的に試し、最良の解を採用する — 大域最適化の単純だが実用的な方法。より高度な手法は探索の順序をより体系立てて行う。

その他の落とし穴と実務のコツ

非線形性は制約より目的関数へ

自動代入オプションを使い、可能なら非線形性を目的関数側に集める

変数のスケールを揃える

値がせいぜい2〜3桁しか違わないよう単位を選ぶ。桁違いはソルバーを誤誘導する

現実的な上下限を課す

定常点・符号の取り違え・偽の実行不可能宣言を防ぎ、成功の見込みを高める

解けたら上下限を確認

「安全のための」上下限に解が達していないか確認し、達していれば緩めて解き直す

まとめ

1. **滑らかさが鍵**
   非線形計画問題の対象は連続かつ勾配が一意に定まる「滑らか」な関数に限られる
2. **3つの発生源**
   仮定を外す・望みの効果を構成する・本質的な物理過程という3経路で非線形性が生じる
3. **初期値と自動代入**
   良い初期値はソルバーの成否を左右し、定義変数は代入で除去してモデルを整理できる
4. **滑らかさを壊さない書き方**
   abs・if_then_else の乱用に注意し、対称的な if や近似で滑らかさを保つ
5. **局所最適 ≠ 大域最適**
   値域違反や複数の局所最適解に注意し、複数の出発点や適切な上下限で対処する

次章：第19章へ続く



AMPL：数理計画のためのモデリング言語

第19章　相補性問題

complements 演算子で経済均衡・ゲーム理論の相補性条件をモデル化する

目 本章の内容

19.1 相補性の発生源

経済均衡モデル、生産経済学の相補性形式、上下限・価格依存需要への拡張、他の応用

19.2 相補性制約の形式

complements 演算子の一般定義（単一不等式の対／二重不等式と式）

19.3 相補性制約を扱う

関連する解の値（接尾辞）、前処理（presolve）、制約の総称的な同義語

相補性問題とは

19.1　相補性の発生源

- さまざまな物理的・経済的現象は、2つの不等式制約が「相補的」——少なくとも一方が等号で成り立つ——であることを要求する形で最も自然にモデル化される。
- 最適化は相補性の特殊な場合とみなせる。線形・非線形計画の標準的な最適性条件そのものが相補性問題である。
- しかし他の相補性問題は最適化から生じるとは限らず、最適化問題として便利に定式化できないこともある。
- **AMPL の complements 演算子は、制約宣言の中で相補性条件を直接記述できるようにする。**

相補性条件

$$x \geq 0$$

complements

$$f(x) \geq 0$$

→ 両方成立、かつ少なくとも一方は等号で成立

x = 0 または f(x) = 0（あるいは両方）

生産経済学の最小費用モデル

19.1　生産経済学の相補性モデル

- 決定変数 Level[j]：活動 j の生産水準
- 目的関数：総生産費用 Total_Cost の最小化
- 制約 Demand：各製品の総産出が需要を満たす

図19-1　econmin.mod（生産費用最小化モデル）

```
set PROD;  set ACT;
param cost {ACT} > 0;
param demand {PROD} >= 0;
param io {PROD,ACT} >= 0;
var Level {j in ACT} >= 0;

minimize Total_Cost:
   sum {j in ACT} cost[j]*Level[j];
subject to Demand {i in PROD}:
   sum {j in ACT} io[i,j]*Level[j]
      >= demand[i];
```

実行結果（CPLEXで solve）

```
ampl: model econmin.mod;
ampl: data econ.dat;
ampl: solve;
CPLEX 8.0.0: optimal solution;
   objective 6808640.553
ampl: display Level;
Level [*] :=
P1a  1555.3   P3   147.465
P3c  1889.4   (他は 0)
```

双対値（制約 i の「価格」）

```
ampl: display Demand.dual;
AA1 16.7051   AC1  5.44585
BC1 57.818    BC2  0
NA2 0         NA3  0
```

最適化から均衡へ：2つの要件

19.1　変数 Level[j] に加え変数 Price[i] を導入

① 価格と需給の相補性

各製品について、総産出は需要を満たさなければならず、価格は非負でなければならない。生産が需要を上回るところでは価格はゼロ、価格が正であれば産出は需要に等しい。

② 水準と価値の相補性

各活動について、産出の総価値は単位費用を超えられない。費用が総価値を上回るところでは活動はゼロ、活動が正であれば総価値は費用に等しい。

この2組の相補性条件をまとめると「正方」相補性問題になる（変数数＝相補性関係数）。

① complements 構文：価格と需給

各製品 *i* について *Pri_Compl* 制約を定義

AMPL 記述

```
subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j] * Level[j] >= demand[i];
```

- 2つの不等式が complements で結ばれるとき、両方が成り立ち、少なくとも一方は等号で成り立つ。
- この例は集合 PROD で添字づけられているため、製品ごとにこの相補性関係が設定される。

② complements 構文：水準と総価値

各活動j についてLev_Compl 制約を定義

AMPL 記述

```
subject to Lev_Compl {j in ACT}:
   Level[j] >= 0 complements
      sum {i in PROD} Price[i] * io[i,j] <= cost[j];
```

活動 j の1単位あたりの総価値 = sum{i in PROD} Price[i]*io[i,j] が費用 cost[j] を超えない。

正方（square）相補性問題

変数の数 ＝ 相補性関係の数（＝活動数＋製品数）

均衡モデルの実行（PATH ソルバー）

図19-3　*econ.mod*

econ.mod（Pri_Compl と Lev_Compl）

```
var Price {i in PROD};
var Level {j in ACT};

subject to Pri_Compl {i in PROD}:
   Price[i] >= 0 complements
      sum {j in ACT} io[i,j]*Level[j]
         >= demand[i];

subject to Lev_Compl {j in ACT}:
   Level[j] >= 0 complements
      sum {i in PROD} Price[i]*io[i,j]
         <= cost[j];
```

実行結果

```
ampl: model econ.mod;
ampl: data econ.dat;
ampl: option solver path;
ampl: solve;
Path v4.5: Solution found.
7 iterations; 33 pivots.

ampl: display
   sum{j in ACT} cost[j]*Level[j];
  = 6808640

→ LP最小費用解と一致。
Price は Demand.dual と一致。
```

上下限をもつ変数への拡張

level_min[j] <= Level[j] <= level_max[j]

混合相補性条件（二重不等式）

```
subject to Lev_Compl {j in ACT}:
   level_min[j] <= Level[j] <= level_max[j] complements
      cost[j] - sum {i in PROD} Price[i] * io[i,j];
```

費用 ＞ 総価値	水準は下限に達する（level_min）
費用 ＝ 総価値	水準は上下限の間にある
費用 ＜ 総価値	水準は上限に達する（level_max）

上下限付きモデルの実行結果

図19-4　econ2.mod を PATH で解く

実行結果（display level_min, Level, level_max）

```
ampl: model econ2.mod;  ampl: data econ2.dat;
ampl: option solver path;
ampl: solve;
Path v4.5: Solution found.  9 iterations; 8 pivots.

ampl: display level_min, Level, level_max;
:   level_min Level level_max :=
P1      240      240    1000
P1a     270     1000    1000
P2      220      220    1000
P2a     260      680    1000
P2b     200      200    1000
P3      260      260    1000
P3c     220     1000    1000
P4      240      240    1000
;
```

結果は、図19-1の変数に同じ上下限を課したLPの解と一致する。

価格に依存する需要への拡張

19.1　LP に等価でない相補性問題

- 需要を固定とせず、価格の減少関数とみなす方が現実的。
- 線形の場合：demzero[i] − demrate[i]*Price[i]
- **Price[i] が complements の両側に現れるため、等価な LP は存在しない。**

図19-5　econnl.mod（抜粋）

```
subject to Pri_Compl {i in PROD}:
  Price[i] >= 0 complements
   sum{j in ACT} io[i,j]*Level[j] >=
     demzero[i] - demrate[i]*Price[i];
```

実行結果（PATH）

```
ampl: model econnl.mod;
ampl: data econnl.dat;
ampl: option solver path;
ampl: solve;
Path v4.5: Solution found.
11 iterations; 11 pivots.

ampl: display Level;
Level [*] :=
  P1   240    P2a  260
P1a   710.156 P2b  200
  P2   220    P3   260
P3c   939.063  P4  240
;
```

→ 需給均衡が生産水準を押し下げる。

他の相補性モデルと応用

19.1　経済均衡以外の相補性問題

MPEC（均衡制約付き数理計画）

目的関数を追加し、等式・不等式・相補性制約を混在させる。解法はまだ実験的段階。

物理系の均衡

力の均衡条件のモデル。相補性制約は2物体の接触／非接触の離散化を表す。

ゲーム理論（ナッシュ均衡）

バイマトリクスゲームのナッシュ均衡は相補性条件で特徴づけられる（各手の確率と期待利得）。

相補性制約の一般形式

19.2　complements 演算子の構文

① 単一不等式どうし

```
single-inequality complements single-inequality ;
```

両方の不等式が成り立ち、少なくとも一方が等号で成り立つときに満たされる。

② 二重不等式と式（混合相補性）

```
double-inequality complements expression ;
expression complements double-inequality ;
```

- 下側 <= または右側 >= が等号 → 式は 0 以上
- 上側 <= または左側 >= が等号 → 式は 0 以下
- どちらの側も等号でない → 式は 0 に等しい

解に関連する値：接尾辞

19.3　cname.Lsuf ／ cname.Rsuf ／ cname.slack

- cname.Lsuf と cname.Rsuf は complements の左オペランド・右オペランドに対応。
- cname.slack は、単一不等式の対では Lslack と Rslack の小さい方。両方満たされていれば非負、厳密に相補性が成り立てば 0。
- 接尾辞なしで cname が現れると cname.slack と解釈される。

display Lev_Compl.Llb, .Lbody, .Rbody, .Rslack

```
ampl: display Lev_Compl.Llb,
ampl?  Lev_Compl.Lbody,
ampl?  Lev_Compl.Rbody,
ampl?  Lev_Compl.Rslack;
:  Llb   Lbody   Rbody   Rslack :=
P1   240    240  1392.86  Infinity
P1a  270   1000  -824.286 Infinity
P2   220    220   264.286 Infinity
P2a  260    680  5.00e-12 Infinity
;
```

前処理（presolve）による簡単化

19.3　相補性制約特有の簡単化

option show_stats 1 の出力

```
Presolve eliminates 16 constraints
   and 2 variables.
Presolve resolves 2 of 14
   complementarity conditions.
Adjusted problem:
12 variables, all linear
12 constraints, all linear
12 complementarity conditions
```

28
_ncons

14
_nccons

12
_sncons

12
_snccons

需要が最小産出量を常に下回らない製品（NA2・NA3）では Price を 0 に固定し、問題から除去できる。

制約の総称的な同義語

19.3　con → ccon への置き換え

モデル作成者の視点（前処理前）

```
_nccons    相補性制約の数
_cconname  相補性制約の名前
_ccon      相補性制約の同義語

各相補性制約 cname は
cname.L と cname.R の2つの
通常制約に展開される。
```

ソルバーの視点（前処理後・標準形）

```
expr complements
   lbound <= var <= ubound
expr <= 0 complements var <= ubound
expr >= 0 complements lbound <= var

solexpand で送信形を確認できる
```

expand Pri_Compl['AA1'] の結果（例）

```
subject to Pri_Compl['AA1']:
    Price['AA1'] >= 0
 complements
    60*Level['P1'] + 8*Level['P1a'] + ... >= 70000;

→ _con[1] = "Pri_Compl['AA1'].L"、_con[2] = "...R"
```

✓ まとめ

1. 相補性条件は「2つの不等式のうち少なくとも一方が等号で成立」を表す自然なモデル化手段であり、最適化の最適性条件もその特殊例である。

2. AMPL の complements 演算子は、単一不等式どうし、あるいは二重不等式と式という2つの構文で相補性制約を宣言できる。

3. 経済均衡モデルでは、価格と需給、活動水準と総価値のあいだの相補性が「正方」相補性問題を構成し、PATH ソルバーで解ける。

4. 上下限をもつ変数や価格依存の需要関数への拡張により、LP に等価でない相補性問題も自然に表現できる。

5. 接尾辞（.Lbody, .Rbody, .slack）、前処理、総称的な同義語（_ccon 等）により、相補性制約も通常の制約と同様に扱える。

次章：第20章へ続く

Z

AMPL：数理計画のためのモデリング言語

第20章　整数線形計画問題

整数変数と0-1（二値）変数で、分割できない実体や論理条件をモデル化する

目 本章の内容

20.1 整数変数

var 宣言に integer キーワードを加える。飲食モデルを例に、LP解と整数解の違いを見る

20.2 0-1変数と論理条件

binary 変数で固定費用・最小輸送量・カーディナリティ制限などの論理条件を表現する

20.3 整数計画法における実務上の考慮事項

整数計画問題の難しさ、変数を絞る工夫、緩和をタイトにする再定式化

整数計画問題とは

整数解が自然に得られる場合と、明示的に強制すべき場合

LPがたまたま整数解を返す場合

第15.5節のネットワーク線形計画問題では整数最適解が保証される。変数の数が制約の数を大きく上回る問題では、多くの変数が上下限にとどまる「極」解が得られ、たまたま整数値になりやすい（図16-4・16-5のスケジューリング問題）。

整数性を明示的に強制すべき場合

そうした保証がない状況は数多く残る。整数計画法ソルバーは線形計画法ソルバーよりはるかに難しい問題に直面し、より多くの計算時間・メモリと、定式化やオプション選択における利用者の助けを必要とする。

本章ではまず整数変数の宣言を説明し、次に論理条件をモデル化する*0-1変数の使い方を紹介する。*

整数変数の宣言

20.1　var 宣言に integer キーワードを加える

- var 宣言の限定句に integer を加えることで、宣言する変数を整数値に制限できる。
- 例：第2.3節の飲食モデルの元のLP最適解では、BEEF が 5.36061、SPG が9.30605という端数の購入量になった。
- 食品を整数量で購入したいなら、var 宣言（図2-1）に integer を加える。

var 宣言に integer を追加（dieti.mod）

```
var Buy {j in FOOD} integer
   >= f_min[j], <= f_max[j];
```

元のLP最適解（display Buy 抜粋）

```
ampl: solve;
MINOS 5.5: optimal solution found.
13 iterations, objective 118.0594032
ampl: display Buy;
Buy [*] :=
BEEF   5.36061    MCH  10
  CHK  2          MTL  10
FISH   2          SPG   9.30605
  HAM 10          TUR   2
;
```

整数解を求める

MINOSは整数性を無視する → 整数対応のソルバーに切り替える

MINOSは整数制約を無視する

```
ampl: model dieti.mod;
ampl: data diet2a.dat;
ampl: solve;
MINOS 5.5: ignoring integrality
   of 8 variables.
13 iterations, objective 118.0594032
```

CPLEXに切り替えて解く

```
ampl: option solver cplex;
ampl: solve;
CPLEX 8.0.0: optimal integer
   solution; objective 119.3
11 MIP simplex iterations
1 branch-and-bound nodes
```

最小費用　118.06 ドル（LP）　→　119.30 ドル（整数）

整数性は追加の制約であるため、目的関数はより不利な方向にしか動かない。BEEFは5.36061→9、SPGは9.30605→7、HAMは10→8に変化。

非整数最適解を最も近い整数値に丸めるだけでは、整数最適解を常に導き出せるわけではない。

B 0-1変数と論理条件

20.2　マルチコモディティ輸送モデルに戻る

- 値として0と1だけをとる変数は整数変数の特別な場合。「0-1」変数、あるいは「二値」変数と呼ぶ。
- 目的関数や制約に巧みに組み込むことで、線形制約だけでは実用的な形で記述できないさまざまな論理条件を指定できる。
- 図4-1のマルチコモディティ輸送モデルに戻る。Trans[i,j,p] は発地 i から着地 j への製品 p の輸送トン数。
- **純粋なLPでは多くの発着対の間で少量の輸送を許してしまう → 0-1変数でこれを抑止する方法を導入する。**

LP最適解（線形計画）

199,500

総変動輸送費用（ドル）

41 iterations で最適解に到達。マルチコモディティ上限 625 に達する経路が6つ、100トン以下の少量輸送も残る。

固定費用の導入

経路ごとの固定費用を目的関数に加える

パラメータと0-1変数の宣言

```
param vcost {ORIG,DEST,PROD} >= 0;
   # variable cost on routes
param fcost {ORIG,DEST} > 0;
   # fixed cost on routes

var Use {ORIG,DEST} binary;
   # =1 only for routes used
```

目的関数（線形の式として書ける）

```
minimize Total_Cost:
   sum {i in ORIG, j in DEST,
        p in PROD}
      vcost[i,j,p] * Trans[i,j,p]
 + sum {i in ORIG, j in DEST}
      fcost[i,j] * Use[i,j];
```

- Use[i,j] は下限0・上限1の整数と言ってもよいし、同じことだがキーワード binary を使ってもよい。
- **連続（非整数）変数と整数変数を組み合わせるため、いわゆる「混合整数」計画問題になる。**

Use と Trans を結びつける制約

Multi制約を修正してUse 変数を組み込む

Multi制約の修正

```
subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];
```

Use[i,j] = 0	sum Trans[i,j,p] <= 0 → 輸送量は0でなければならない（非負変数の和のため）
Use[i,j] = 1	sum Trans[i,j,p] <= limit[i,j] → 以前と同じマルチコモディティ上限

fcost[i,j] が正である限り、最適解ではUse[i,j] は輸送量が正であるとき、かつそのときに限り 1 になる。

完全なモデル：multmip1.mod

図20-1a　固定費用付きマルチコモディティモデル

```
set ORIG;  set DEST;  set PROD;
param supply {ORIG,PROD} >= 0;   param demand {DEST,PROD} >= 0;
param limit {ORIG,DEST} >= 0;    param vcost {ORIG,DEST,PROD} >= 0;
var Trans {ORIG,DEST,PROD} >= 0; # units to be shipped
param fcost {ORIG,DEST} >= 0;    # fixed cost for using a route
var Use {ORIG,DEST} binary;      # = 1 only for routes used

minimize Total_Cost:
   sum {i in ORIG, j in DEST, p in PROD} vcost[i,j,p] * Trans[i,j,p]
 + sum {i in ORIG, j in DEST} fcost[i,j] * Use[i,j];
subject to Supply {i in ORIG, p in PROD}:
   sum {j in DEST} Trans[i,j,p] = supply[i,p];
subject to Demand {j in DEST, p in PROD}:
   sum {i in ORIG} Trans[i,j,p] = demand[j,p];
subject to Multi {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] <= limit[i,j] * Use[i,j];
```

固定費用モデルの実行結果

図20-1bのデータ（multmip1.dat）を用いて解く

199,500	223,504	229,850
LP （固定費用なし）	整数性を無視 （MINOS、Useが端数）	真の整数最適解 （CPLEX）

CPLEXによる整数解の探索

```
ampl: option solver cplex; solve;
CPLEX 8.0.0: optimal integer solution; objective 229850
295 MIP simplex iterations   19 branch-and-bound nodes
```

整数制約を課すと総費用は223,504ドルから229,850ドルへ増加する一方、望んでいた通り使われない経路の数は0から7に増える。

≥ ゼロまたは最小量の制限

Min_Ship 制約 — 輸送するなら minload 以上

AMPLは制約の中で論理演算子 or を受け付けない

```
subject to Min_Ship {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] = 0 or
   sum {p in PROD} Trans[i,j,p]
      >= minload;          # NOT ALLOWED
```

Use[i,j] を使った正しい定式化

```
subject to Min_Ship {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p]
      >= minload * Use[i,j];
```

Use[i,j] が1なら最小輸送量制約になり、Use[i,j] が0なら 0 >= 0 に帰着して何の効果ももたない。

minload = 375 で解くと、目的関数は 233,150 ドル（+1.4%）

```
CPLEX 8.0.0: optimal integer solution; 279 MIP simplex iterations, 17 branch-and-bound nodes
```

使われない経路数は依然7つだが、同じ経路ではない。最小輸送量を満たすため解の実質的な組み換えが必要になった。

カーディナリティの制限

Max_Serve 制約 — 発地ごとの配送先数を制限

変数で定義される集合は制約の card に使えない

```
subject to Max_Serve {i in ORIG}:
   card {j in DEST:
      sum {p in PROD} Trans[i,j,p]
         > 0} <= maxserve;    # NOT ALLOWED
```

Use[i,j] を使った正しい定式化

```
subject to Max_Serve {i in ORIG}:
   sum {j in DEST} Use[i,j]
      <= maxserve;
```

sum {j in DEST} Use[i,j] は発地 i が配送する着地の数を表す。maxserve = 5 として解く。

固定費用のみ	229,850 ドル ／ 未使用経路 7
+ 最小輸送量（minload = 375）	233,150 ドル （+1.4%）
+ カーディナリティ（maxserve = 5）	235,625 ドル （+1.1%）

整数計画法の実務上の考慮事項

20.3　整数計画法における実務上の考慮事項

整数計画問題ははるかに解くのが難しい

同じ規模のLPよりずっと難しい。反復回数の目安：線形版の41回に対し、混合整数版は約280〜400回。計算時間も同様に増大する。

整数変数は少なく、上下限は厳しく

問題規模が増すと整数計画法の難しさは急速に増大する。連続な線形・ネットワーク定式化で十分でないか、まず検討すべきである。

緩和がタイトになるよう再定式化する

個々の Trans[i,j,p] を Use[i,j] に連動させる制約を追加すると、LP緩和での Use[i,j] が1に近づき、整数最適解を速く見つける助けになる。

再定式化の例：Avail制約（緩和をタイトにする）

```
subject to Avail {i in ORIG, j in DEST, p in PROD}:
   Trans[i,j,p] <= min(supply[i,p],demand[j,p]) * Use[i,j];
```

✓ まとめ

1. 整数変数は var 宣言に integer を、0-1（二値）変数は binary を加えて宣言する。パッケージや人など分割できない実体や、論理条件のモデル化に使われる。

2. LPソルバーがたまたま整数解を返すこともあるが（ネットワーク構造、極解など）、保証がない場合は整数計画法ソルバー（CPLEXなど）に明示的に切り替える必要がある。

3. 0-1変数 Use[i,j] を輸送量に連動させる制約（Trans <= limit*Use）により、固定費用・最小輸送量・カーディナリティ制限といった論理条件を線形制約として表現できる。

4. 整数制約は目的関数をより不利な方向にしか動かせず、非整数解を丸めるだけでは整数最適解を得られない。

5. 整数計画問題はLPよりはるかに解くのが難しい。少ない整数変数・厳しい上下限・緩和をタイトにする再定式化・実験的な試行錯誤が成功への鍵となる。

次章へ続く

AMPL：数理計画のためのモデリング言語

$A

付録A　AMPLリファレンスマニュアル

クイックリファレンス　—　字句規則から起動オプションまで、全23節の構文早見

23	14	全22章
節	表 (A-1〜A-14)	シリーズ最終回

Fourer, Gay, Kernighan 著　日本語版

目 付録Aの構成 —23節を5カテゴリーで整理

アジェンダ

本編の概念解説に対し、付録A*は逐条的な文法リファレンス。カテゴリー別に要点を凝縮する。*

字 字句・式編

4節

字句規則 / 集合のメンバー / 添字式と下付き添字 / 式

宣 宣言編

7節

モデル実体の宣言 / 集合 / パラメータ / 変数 / 制約 / 目的関数 / 接尾辞記法

デ データ編

2節

標準データ形式 / データベースへのアクセスと表

コ コマンド編

7節

コマンド概観 / リダイレクト / 印字と表示 / データ読込 / モデル化コマンド / モデル検査 / 制御フロー

環 環境・拡張編

3節

計算機環境 / インポートされる関数 / AMPLの起動

字句規則

名前・数値定数・リテラル・コメント・予約語・演算子の同義語

A.1

- 名前: Unicode文字・数字・下線からなる英数字。大文字小文字を区別
- 数値定数: 標準の科学的記法。指数部は d/D/e/E のいずれでも可
- リテラル: ' または " で区切る文字列。区切り文字はそれを二重にしてエスケープ
- 文はセミコロンで終わる。空白(スペース・タブ・改行)は自由に置ける
- コメント: # から行末まで、または /* … */ (入れ子不可)
- 予約語(約44語)は他の用途に使えない。それ以外のキーワードは再定義可能

リテラル・コメント

```
1.23D-45            # 科学的記法(d/D/e/Eは同義)
'x''y'   ==  x'y    # 引用符の二重化でエスケープ
set prod;           # 事前定義語 prod を再定義
# コメントは行末まで
/* 複数行のコメント */
```

演算子の同義語 (左が推奨形)

`^  **`	べき乗
`=  ==`	等号
`<>  !=`	不等号
`and  &&`	論理積
`not  !`	論理否定
`or  ||`	論理和
`prod  product`	総乗の縮約演算子

予約語の例

`if`	`then`	`else`	`while`
`sum`	`forall`	`exists`	`in`
`within`	`union`	`integer`	`symbolic`

集合のメンバー / 添字式と下付き添字

A.2–A.3

集合はメンバーの重複しない順序付きリスト。添字式はダミーメンバーで反復範囲を指定する

集合リテラル

```
{"a","b","c"}                  # 1次元
{1,2,3,4,5,6,7,8,9}
{("a",2),("a",3),("b",5)}      # 多次元(タプル)
```

下付き添字 と 添字式の構文

```
cost[j, p[k]+1, "O+"]
indexing: { sexpr-list [ : lexpr ] }
sexpr-list: dummy-member in sexpr , ...
```

よく使う添字式パターン

パターン集

```
{A}                            # Aの全要素
{i in A, j in B}               # A×B の全ペア
{i in A, C[i]}                 # Aと C[i] の組
{i in A: p[i] > 0}             # such that (条件選択)
{i in A, (j,k) in D: i<=j}     # 多次元集合との組合せ

# ダミー添字の有効範囲は宣言/式の終わりまで
```

集合は 0 個以上のメンバーからなり、各メンバーは同じ成分数(=次元)をもつ。
1 と 0.01E2 のように浮動小数点として同じ値に丸まる数値メンバーは同一視される。

式 (1) — 演算子と優先順位

A.4

算術式 *expr*・*論理式 lexpr*・*集合式 sexpr* の3種類。表A-1は16段階の優先順位を規定

3種類の式

expr (A)	算術式。変数を含んでよい
cexpr	変数を含まない「定数式」
lexpr (L)	真偽を返す論理式
sexpr (S)	集合を返す式

式の例

```
sum {i in Prod} cost[i] * Make[i]
if x > 0 then 1 else 0
A union B     A diff B     A inter B
1 .. 10 by 2                # 等差数列の集合
setof {i in A} (i, i^2)     # 集合構成子
```

優先順位(抜粋・低い順)

1	**if-then-else**	A/S、else省略時は 0
2–4	**or/exists/and**	論理和・論理縮約・論理積
5	**比較演算子**	< <= = <> >= >
8–10	**union/inter/cross**	集合演算
12	**+ - less**	a less b = max(a-b,0)
13	**sum prod min max**	算術縮約演算子
16	**^ ****	べき乗(右結合)

関 式 (2) — 組込み関数と区分的線形項

A.4

算術・文字列・乱数の組込み関数(表A-2〜A-4)、および区分的線形項の構文

算術関数

abs	sqrt	exp	log	log10
ceil	floor	round(x,n)	trunc	min/max

文字列・正規表現関数

s & t	length	substr	sprintf	match
sub	gsub	num	num0	char

乱数生成関数

Uniform01()	Uniform(m,n)	Normal(μ,σ)	Poisson(μ)
Exponential()	Gamma(a)	Beta(a,b)	Irand224()

区分的線形項

```
<< breakpoints ; slopes >> var
<< 0,10 ; -1,0,1 >> x     # 折れ点;傾き
```

正規表現の主なメタ文字

.	任意の1文字
[a-z0-9]	文字クラス
r* r+ r?	0回以上・1回以上・0または1回
^r r$	先頭一致・末尾一致
r1|r2	選択(または)

集 モデル実体の宣言 / 集合の宣言

A.5–A.6

全実体は共通の宣言形式に従う。集合は *dimen / within / = / default の属性をもつ*

共通形式

```
entity name alias_opt indexing_opt body_opt ;
entity: set param var arc minimize maximize
        subject to  node
```

集合宣言の属性

dimen n	次元を明示
within sexpr	部分集合であることを要求(複数可)
= sexpr	値を確定(データ節で変更不可)
default sexpr	データ未指定時の既定値

順序付き集合 ordered / circular

```
set S ordered [ by [reversed] sexpr ];
set S circular [ by [reversed] sexpr ];

next(e,S,k)  prev(e,S,k)  first(S)  last(S)
member(j,S)  ord(e,S)  card(S)
```

推移閉包の例

```
set nodes;
set arcs within nodes cross nodes;
param n := card(nodes)-1;   # card(s)=要素数
set reach = ...;            # 再帰的な集合定義
```

P パラメータの宣言

A.7

param name alias_opt indexing_opt attributes_opt ; / check文とInfinity

属性

binary / integer	0,1に制限 / 整数に制限
symbolic	文字列も許容(比較演算子属性は不可)
relop expr	< <= = >= > による制約検査
in sexpr	所属する集合を検査
= expr / default expr	値を確定 / 既定値(互いに排他)

再帰的な定義 と Infinity

```
param comb {n in 0..N, k in 0..n} =
  if k=0 or k=n then 1
  else comb[n-1,k-1] + comb[n-1,k];
# 以前に計算された値だけを参照すれば再帰可
```

check文

```
check [ indexing_opt : ] lexpr ;
check {i in A}: p[i] >= 0;   # solve/write等で評価
```

Infinity

```
param Ub{A} default Infinity;
param Lb{A} default -Infinity;
var V {i in A} >= Lb[i], <= Ub[i];
```

V 変数の宣言

A.8

var name alias_opt indexing_opt attributes_opt ; / 定義変数と弧(arc)

主な属性

`binary / integer`	0,1に制限 / 整数に制限
`>= expr / <= expr`	下限 / 上限
`:= expr / default`	初期値 / 既定初期値(排他)
`= expr`	定義変数(他の属性と併用不可)
`coeff / obj / cover`	列方向の係数生成
`suffix sufname expr`	接尾辞の初期値

定義変数のポイント

- option substout=1 で、定義制約の左辺変数を共通部分式として除去
- 整数性・上下限などの制約を持つ変数は定義変数になれない
- option linelim=1(既定)で線形の定義変数を特別扱い

```
var x {J} >= 0, <= 100, integer;
var w = 2*x[1] + x[2];   # 定義変数(名前付き部分式)
```

arc宣言

```
arc F {(i,j) in ARCS} >= 0, <= cap[i,j],
    from node[i], to node[j];
# from/to は node制約に係数を寄与
```

制 制約の宣言 / 目的関数の宣言

A.9–A.10

[subject to] name : expr ; / minimize|maximize name : expr ; / 相補性制約

制約の形式

```
[subject to] name indexing_opt
  [:= initial_dual] [default d] [: expr] ;
expr: e<=e | e=e | e>=e | c<=e<=c | c>=e>=c
```

目的関数の形式

```
minimize Cost: sum {j in J} cost[j]*x[j];
maximize Profit: to_come;   # 列方向の係数生成
```

ノード制約 (net_in / net_out)

```
node Bal {i in NODES}: net_in = net_out;
```

相補性制約

```
name : constr1 complements constr2 ;

expr4 complements lb <= var <= ub
```

- constr1・constr2 は合計2個の不等式演算子を含む(=は2個と数える)
- 両方が成り立ち、少なくとも一方が等式で満たされるとき充足
- 均衡制約付き数理計画(MPEC)の表現に使う。他の制約・目的関数と共存可能
- 同義語 _scvar[i] がどの変数がどの制約に相補するかを示す

尾 補助的な値のための接尾辞記法

name.suffix で変数・制約・目的関数の付随値にアクセス(表A-6〜A-8)

変数の接尾辞(抜粋)

`.lb .ub`	下限・上限
`.val`	現在の値
`.rc`	換算費用
`.status`	ステータス

制約の接尾辞(抜粋)

`.body`	制約本体の値
`.dual`	双対変数
`.slack`	スラック
`.lb .ub`	下限・上限

目的関数の接尾辞

`.val`	目的関数の現在値

```
display x.lb, x.ub, x.rc;
display Balance.dual;
```

suffix宣言(方向)

```
suffix name alias_opt attributes_opt ;
suffix priority IN, integer;
direction: IN | OUT | INOUT | LOCAL   (既定 INOUT)
```

ステータス接尾辞

- .sstatus: ソルバーが返す基底情報 (次のsolveで再利用)
- .astatus: AMPLのdrop/restore・fix/unfixの状態を反映
- .status: .astatus と .sstatus から導かれる総合ステータス

デ 標準データ形式 (1) — 集合のデータ

*data*節はキーワード・リテラル・数・区切り文字からなる書式自由なトークン列

1次元・タプル集合

```
data;
set A := a b c;                 # 1次元(引用符省略可)
set B := "x y" z;               # 空白を含む文字列は引用符
set PAIR := (a,1) (a,2) (b,3);
```

2次元集合(表形式)

```
set PAIR : b1 b2 b3 :=
   a1     +   -   +
   a2     -   +   -  ;
# 行=第1添字, 列=第2添字, + は所属, - は非所属
```

集合データ文の一般形

```
set set-name := set-spec set-spec ... ;
set-template: ( templ-item, ... )
templ-item: object | * | :
```

- テンプレートの * にオブジェクトが左から代入される
- テンプレート省略時は全て * (集合の次元数分)が仮定される
- (tr) を付けると表が転置され、行と列の役割が入れ替わる
- 添字付き集合は成分ごとに別々のデータ文で与える
- データの書式(改行・空白)は人間の可読性のためで、AMPL自体は無視する

標準データ形式 (2) — パラメータのデータ

param param-name := value-list | value-table ; / 複数パラメータの同時定義

1次元(リスト形式)

```
param cost :=
  a1  10
  a2  20 ;
```

複数同時定義

```
param: DEMAND: d cost :=
  chi   200    1.5
  den   150    2.1 ;
# 集合DEMANDと複数パラメータを同時定義
```

```
param p {A} default 0 := a1 5, a2 . ;   # . は欠損値→既定値
```

param表の要点

- テンプレートは丸括弧でなく角括弧 [* , *] で与える
- 2次元表は +/- でなく数値(記号パラメータはリテラル)を格納
- default r 句で、値未指定のパラメータに既定値を一括付与
- 「.」は欠損値。default指定があればその値に解決される
- var name はデータ節では param の同義語(変数の初期値指定に使用)

```
var x {J} := 0;       # 初期値もparamと同じ書式で指定可
```

表 データベースへのアクセスと表

table宣言は外部リレーショナルテーブルの列とAMPLの集合・パラメータを接続する

table宣言の形式

```
table table-name indexing_opt in-out_opt
   string-list_opt :
   key-spec, data-spec, data-spec, ... ;
in-out: IN | OUT | INOUT   (既定 INOUT)
```

使用例

```
table Orders IN "ODBC" (dsn="mydb"):
   [ORD] ORD ~ id, Qty ~ qty;
read table Orders;    # 外部→AMPL
write table Orders;   # AMPL→外部
```

キー列とデータ列

`key-name`	ダミー添字と列名を兼用
`key ~ col`	ダミー添字keyを列colに対応付け
`param ~ col`	AMPL名を外部列名に対応付け
`<- -> <->`	読込/書出/双方向の向き

- 組込みハンドラは .tab / .bit ファイルの読み書きに対応
- DB・スプレッドシート接続には別途ハンドラの導入が必要
- 組込み集合 _HANDLERS が利用可能なハンドラ名を列挙

コマンド言語の概観

A.14

モデルモード / データモードの2モード。option コマンドでオプションを管理(表A-9)

基本フロー

```
model diet.mod;     data diet.dat;
option solver cplex;
option *col*;    # 名前パターンで一括表示
```

- data文でデータモードへ。モデルモードに戻すのは他のコマンド
- include filename でファイルを挿入(model/data命令の省略形あり)
- $opname=現在値、$$opname=既定値(環境から継承 or AMPLの既定)
- option opname evalue; で設定、option opname; で現在値を表示

代表的なコマンド(表A-9より抜粋・全44種)

solve	display	print	printf
let	fix	unfix	drop
restore	expand	show	xref
reset	write	read	option
problem	check	cd	shell

印 入出力のリダイレクト / 印字と表示のコマンド

A.15–A.16

> >> < でファイル入出力を切替。display / print / printf で結果を整形出力

リダイレクト

```
> filename      # 上書き(最初にオープン時)
>> filename     # 追記
< filename      # read コマンド用の入力
close filename;   remove filename;
```

3つの出力コマンド

```
display [indexing:] disparglist redirection ;
print   [indexing:] arglist redirection ;
printf  [indexing:] fmt, arglist redirection ;
```

```
printf "%s: %8.2f\n", j, cost[j] > "out.txt";
```

printf 変換指定(表A-10・抜粋)

%d %i	符号付き10進
%s	文字列
%f %e %g	浮動小数点(固定/指数/自動)
%q %Q	必要時/常に引用符付き文字列
%x %o	16進 / 8進

- display: 数値を既定で有効数字6桁に丸める(display_precision等で変更)
- print: 完全精度。区切り文字は print_separator で変更可
- display_1col, display_width, gutter_width 等が表の体裁を制御

解 データの読み込み / solve・write コマンド

A.17–A.18

read で書式なしデータを取込み。solve でソルバー起動、write で問題を書き出す

read コマンド

```
read [indexing:] arglist redirection ;
read {j in DEST} demand[j] < "demand.txt";
```

solve コマンド

```
solve redirection_opt ;
  1) 一時ファイルへ問題を書出し
  2) option solver の値でソルバーを起動
  3) stub.sol から主変数・双対変数を読込む
```

```
write bstub;    # スタイル文字+スタブでファイル生成
```

solveに関わる主なオプション

`solver`	起動するソルバー名
`relax_integrality`	integer/binary属性を無視
`reset_initial_guesses`	初期推定値を送るか制御
`send_suffixes`	接尾辞値の送受信を制御
`solve_result_num`	ソルバーの戻りコード

更 モデル化コマンド — 変更・固定・問題・データ更新

delete / drop / fix / problem / reset / let で実体やデータを操作する

実体の変更・除去

delete / purge	実体を削除 / 依存物ごと削除
redeclare	実体の宣言を再定義
drop / restore	制約・目的関数をソルバーから除外/復帰
fix / unfix	変数を現在値に固定/解除

```
fix x[1] := 5;
drop Balance[1];
let demand[j] := demand[j]*1.1;
```

問題(problem)とデータ更新

```
problem name indexing_opt : itemlist ;
problem name;      # 切替
problem SubProb: x, Balance;
```

reset data	読込済みデータを破棄
update data	後続データ節での上書きを許可
let name := expr	集合・パラメータ・変数の値を変更

検 モデルの検査

A.19

show / xref / expand / solexpand と組込みの総称的な名前(表A-13)

show namelist	実体の名前・種類を列挙
xref itemlist	依存する実体を表示
expand itemlist	生成された制約・目的関数を展開表示
solexpand	ソルバーから見た展開(前処理後)を表示
check;	全ての check 文を評価

組込みの総称的な名前(抜粋)

_nvars	**_ncons**	**_nobjs**
_VARS	**_CONS**	**_OBJS**
_varname{}	**_conname{}**	**_PARS**

```
show;                  # 全モデル実体を列挙
expand Balance[1];     # 特定制約の展開
```

```
display _nvars, _ncons, _nobjs;
```

制 スクリプトと制御フロー文

if-then-else / for / repeat による条件分岐とループでスクリプトを自動化

A.20

for + if-then-else

```
for {t in TIME} {
  if t <= 6 then
    let cmin[t] := 3;
  else
    let cmin[t] := 4;
}
```

repeat / break / continue

```
repeat loopname_opt while lexpr { cmds } ;
break loopname_opt ;   continue loopname_opt ;
```

構文まとめ

`if lexpr then cmd`	[else cmd] 条件分岐
`for indexing cmd`	添字集合にわたる反復
`repeat {cmds}`	while/until条件付き反復
`break / continue`	ループの中断 / 次の反復へ

- cmd はセミコロン終端の単文、または{ }で囲んだ文の並び
- option single_step n; で1コマンドずつ実行するデバッグモード

環 計算機環境

shell / cd コマンド、組込みの時間計測パラメータ、ログ記録オプション(表A-14)

外部コマンド・終了

```
shell 'command-line' redirection_opt ;
cd new-directory ;
quit;   exit expr_opt;   end;
```

- quit: $outoptが暗示するファイルを書き出さず終了
- exit: quitの同義語。ステータス値を環境に返せる
- log_file が空でなければ標準入力の内容をログに複写
- log_model / log_data で他ファイルのコマンド・データも記録

時間計測パラメータ(抜粋)

`_solve_time`	直近solveのCPU時間
`_total_solve_time`	全solveの累積CPU時間
`_shell_elapsed_time`	直近shellの経過時間
`_ampl_elapsed_time`	AMPLプロセス開始からの経過秒数

```
display _total_solve_time;
```

関 インポートされる関数

function宣言で外部関数を取込む。パイプ関数は別プロセスとして評価される

function宣言

```
function name alias_opt (domain-spec)_opt
  type_opt [pipe litseq_opt [format fmt]] ;
```

使用例

```
function mean2 pipe "awk '{print ($1+$2)/2}'";
display mean2(1,2) + 1;
call myfunc(x, y);   # 戻り値を使わない呼出し
```

ドメイン方向とロード

`IN / OUT / INOUT`	情報の流れる向き(既定IN)
`symbolic / random`	文字列を返す/毎回異なる値を返す
`load / unload / reload`	共有ライブラリの制御
`reset function name`	パイプ関数を閉じて再起動可能に

- domain-spec が集合を指定する引数は、その集合への所属を検査
- 単独の * はドメイン検査を省略する印

起 AMPL の起動

コマンドライン引数とセッション間で設定を記憶するオプション(OPTIONS_IN 等)

コマンドライン例

```
ampl '-?'           # 起動オプション一覧を表示
ampl -s1 model.mod  # 乱数シードを1に指定
ampl -R             # 制限付きサーバーモード
```

セッションを跨ぐオプション記憶

OPTIONS_IN	起動時に読み込むオプションファイル
OPTIONS_INOUT	起動時に読込み、終了時に書出す
OPTIONS_OUT	終了時に現在の設定を書出す

- -v: AMPLのバージョンを印字(option version と同等)
- -ilibs: 最初にロードする関数/tableハンドラのライブラリを指定
- -R: cd/shell禁止、ソルバー名やリダイレクト名を英数字に制限

ア まとめ —本編と付録Aの使い分け

本編(第1〜21章)

- モデリングの考え方・ワークフローを、実例を通じて学ぶ章立て
- 生産・輸送・スケジューリングなど分野別の応用例が中心

付録A(本スライド)

- 正確な文法・キーワード・オプション名を引くための逐条リファレンス
- コーディング中に「あの構文は？」を素早く確認する用途

よく使うキーワード早見表

`set`	`param`	`var`	`subject to`
`minimize`	`maximize`	`sum`	`forall`
`if-then-else`	`in / within`	`display`	`solve`
`let`	`option`	`table`	`function`

全22章シリーズ、完結。

詳細な最新情報は www.ampl.com を参照