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

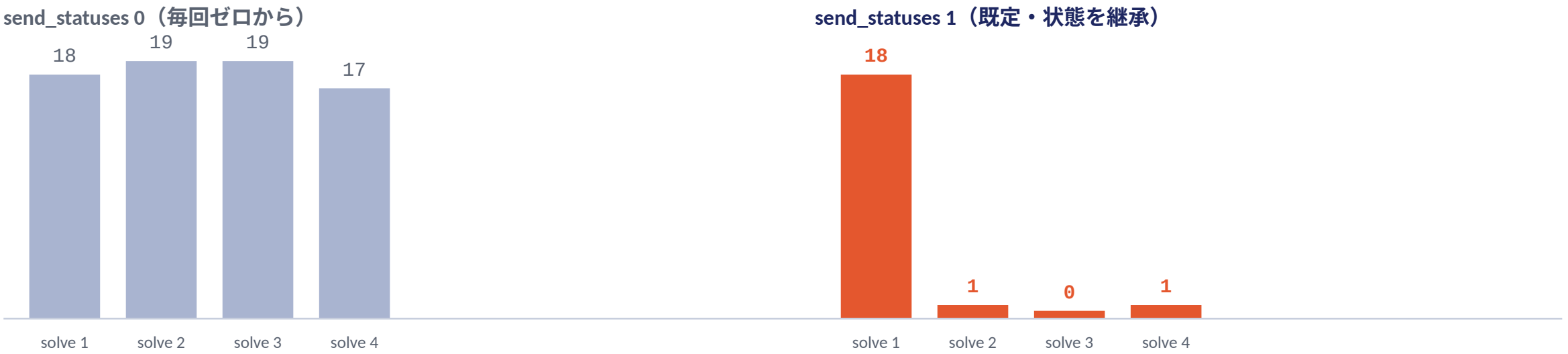
```
ampl: solve;
MINOS 5.5: optimal solution found.

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