C

付録C

AMPL MP 式のリファレンス

MP ベースのソルバーで使える論理・計数・非線形などの式

この章の内容

1 条件演算子

2 論理演算子

3 区分線形関数

4 計数（カウント）演算子

5 比較・相補性・集合への所属

6 二次式・錐・一般の非線形関数

7 書き換えに関する主要なオプション

C.1 条件演算子

条件に変数を含んでよい if-then-else と含意の演算子

条件演算子の意味

書き方	意味
if constr then expr1 [else expr2]	constr が真なら expr1、偽なら expr2（else 省略時は 0）
constr1 ==> constr2 [else constr3]	constr1 が真なら constr2、偽なら constr3 が成り立つ
constr2 <== constr1	constr1 ==> constr2 と同じ
constr1 <==> constr2	両方が真か、両方が偽か、のいずれか

constr1 が *binary_var = 0 / 1* の形なら指示制約になり、*Gurobi*、*CPLEX*、*COPT*、*Xpress* がネイティブに処理する

条件演算子の例

model.mod

```
minimize TotalCost:
   sum {j in JOBS, k in MACHINES}
      if MachineForJob[j] = k then cost[j,k];

subject to Balance {p in PROD, t in TIME}:
   Make[p,t] + (if t = 0 then inv0[p] else Inv[p,t-1])
      = Sell[p,t] + Inv[p,t];

subject to Multi_Min_Ship {i in ORIG, j in DEST}:
   sum {p in PROD} Trans[i,j,p] > 0 ==>
      minload <= sum {p in PROD} Trans[i,j,p] <= limit[i,j];

subject to Least_Use {j in SCHEDS}:
   Use[j] = 1 ==> Work[j] >= least_assign else Work[j] = 0;
```

C.2 論理演算子

or・and・not と、添字付きの exists・forall

論理演算子の意味

書き方	満たされる条件
constr1 or constr2	どちらかが真
constr1 and constr2	両方が真
not constr	constr が偽
exists {添字} constr	少なくとも 1 つの項が真（添字付きの or）
forall {添字} constr	すべての項が真（添字付きの and）

論理演算子の例（1）

model.mod

```
# 選言的なスケジューリング：big-M は不要
subj to No_Conflict {i1 in JOBS, i2 in JOBS: ord(i1) < ord(i2)}:
   Start[i2] >= Start[i1] + t_offset[i1,i2] or
   Start[i1] >= Start[i2] + t_offset[i2,i1];

subject to Least_Use {j in SCHEDS}:
   Work[j] = 0 or
   least_assign <= Work[j] <= max {i in SHIFT_LIST[j]} required[i];

subj to EntRem {t in 1..numTanks}:
   Entry[t] + minTime[t] <= Removal[t] and
   Entry[t] + maxTime[t] >= Removal[t];
```

論理演算子の例（2）exists と forall

model.mod

```
# exists を使った固定費用の表現
minimize Total_Cost:
   sum {p in PRODUCTS, (i,j) in ARCS}
      var_cost[p,i,j] * Flow[p,i,j] +
   sum {(i,j) in ARCS}
      if exists {p in PRODUCTS} Flow[p,i,j] > 0
      then fix_cost[i,j];

# 添字付きの論理演算子
subject to Multi {i in ORIG, j in DEST}:
   forall {p in PROD} Trans[i,j,p] = 0 or
   minload <= sum {p in PROD} Trans[i,j,p] <= limit[i,j];
```

- いずれかの品目が枝を流れれば固定費用がかかる
- 全品目の輸送量が 0 か、合計が範囲内か

C.3

区分線形関数

abs・min・max と << 折れ点; 傾き >> の演算子

abs, min, max

model.mod

```
maximize Total_Profit:
   sum {p in PROD, t in 1..T}
      revenue[p,t]*Sell[p,t] -
   sum {t in 1..T}
      time_penalty[t] * abs(Use[t] - avail_min[t]);

minimize Max_Cost:
   max {i in PEOPLE}
      sum {j in PROJECTS} cost[i,j] * Assign[i,j];

maximize WeightSum:
   sum {t in TRAJ}
      max {n in NODE} weight[t,n] * Use[n];
```

- 対応するソルバーはネイティブに処理
  - Gurobi の ABS/MIN/MAX の一般制約
- それ以外では自動的に線形化される

区分線形の演算子（折れ点と傾き）

model.mod

```
option pl_linearize 0;   # MP ベースのソルバーでは必ず設定

maximize Total_Profit:
   sum {p in PROD, t in 1..T} (revenue[p,t]*Sell[p,t] -
      prodcost[p]*Make[p,t] -
      <<0; -backcost[p],invcost[p]>> Inv[p,t]) -
   sum {t in 1..T} <<avail_min[t]; 0,time_penalty[t]>> Use[t];

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];
```

C.4

計数（カウント）演算子

条件を満たす要素の個数や、値が相異なることを表す

計数演算子の意味

書き方	意味
count {添字} constr	constr が満たされる要素の個数
atleast k {添字} constr	constr が少なくとも k 個の要素で満たされる
atmost k {添字} constr	constr が高々 k 個の要素で満たされる
exactly k {添字} constr	constr がちょうど k 個の要素で満たされる
numberof expr in (expr のリスト)	リストの中で expr に等しい項目の個数
alldiff {添字} expr	すべての値が相異なる

計数演算子の例

model.mod

```
subject to Min_Serve {i in ORIG}:
    count {j in DEST} (Ship[i,j] >= minload) >= minserve;
# 同じ制約を atleast で
subject to Min_Serve {i in ORIG}:
    atleast minserve {j in DEST} (Ship[i,j] >= minload);

subj to CapacityOfMachine {k in MACHINES}:
    atmost cap[k] {j in JOBS} (MachineForJob[j] = k);
# 同じ制約を numberof で
subj to CapacityOfMachine {k in MACHINES}:
    numberof k in ({j in JOBS} MachineForJob[j]) <= cap[k];

subject to OnePersonPerPosition:
   alldiff {i in 1..nPeople} Pos[i];

subject to Regions {I in 1..9 by 3, J in 1..9 by 3}:
   alldiff {i in I..I+2, j in J..J+2} X[i,j];
```

C.5

比較・相補性・集合への所属

C.5 比較演算子、C.6 相補性の演算子、C.7 集合への所属の演算子

3 つの演算子

いずれも書き換えは MP が担う

1 比較演算子 !=

制約の中でそのまま使える

2 complements

constr1 complements constr2：両方が満たされ、少なくとも一方は等号で成り立つ

3 集合への所属

var X in 集合の式; 変数は指定した集合の中の値を取る

比較・相補性・集合への所属の例

model.mod

```
# 比較演算子
subject to Different_Colors {(c1,c2) in Neighbors}:
   Color[c1] != Color[c2];

# 相補性の演算子
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];

# 集合への所属の演算子
var Buy {f in FOODS} in {0, 10, 30, 45, 55};
var Ship {(i,j) in ARCS}
   in {0} union interval[min_ship, capacity[i,j]];
```

- 隣接する地域を異なる色に塗る
- 価格と超過供給の相補性
- Buy はとびとびの値、Ship は 0 または区間内

C.8

二次式・錐・非線形と書き換えオプション

C.8 二次式・べき乗、C.9 錐、C.10 一般の非線形関数、C.11 オプション

二次式・べき乗と錐最適化

model.mod

```
minimize Portfolio_Variance:
   sum {i in ASSETS, j in ASSETS}
      covariance[i,j] * w[i] * w[j];

# 標準的な二次錐
s.t. cone: x[0] >= sqrt(x[1]^2 + x[2]^2 + x[3]^2);

# 回転二次錐 (x[0], x[1] >= 0)
s.t. rotated: 2*x[0]*x[1] >= x[2]^2 + x[3]^2;

# 指数錐（Mosek がネイティブに扱う）
s.t. T1: 1 + b*w >= exp(q1);
```

- 凸 QP はソルバーがネイティブに処理
- 非凸の二次式は Gurobi、Xpress、SCIP などがそのまま処理
- SOCP は代数的な書き方から認識される

一般の非線形関数

model.mod

```
minimize Chichinadze:
   x[1]^2 - 12*x[1] + 11 + 10*cos(pi*x[1]/2) +
      8*sin(pi*5*x[1]) - exp(-(x[2]-.5)^2/2)/sqrt(5);
```

- exp、log、log10
- sin、cos、tan とその逆関数
- sinh、cosh、tanh とその逆関数
- 指数部に変数を含む ^
- 線形・MIP ソルバーには区分線形で近似
- Gurobi 11 以降は大域的、12 以降はネイティブ

書き換えに関する主要なオプション

オプション	働き
option pl_linearize 0;	区分線形の処理を MP に任せる（MP ベースのソルバーでは推奨）
cvt:mip:eps	狭義不等式の許容誤差（x > 5 は x >= 5 + eps になる）
cvt:quadobj、cvt:quadcon	二次式の書き換えの制御
cvt:socp、cvt:socp2qp	SOCP の書き換えの制御
cvt:compl	相補性の書き換えの制御

まとめ

- ✓ 現行の AMPL のソルバーはすべて MP ベース
- ✓ 条件・論理・計数の式で big-M が不要になる
- ✓ 区分線形は abs・min・max と << ; >> で書く
- ✓ !=、complements、集合への所属も使える
- ✓ 二次・錐・非線形は MP がソルバーに合わせて扱う