GAMS Basics

Stefan Vigerske

stefan@gams.com

Some of the following material is based on the GAMS lecture material by Josef Kallrath.

Outline

Basic Modeling

Dynamic Models

More GAMS

Outline

Basic Modeling

Dynamic Models

More GAMS

Cows & Pigs Example

Variables:

x_1 the number of cows to purchase

x_2 the number of pigs to purchase

Objective:

$$\text{maximize} \quad z = 3x_1 + 2x_2$$

Constraints:

$$x_1 \in \{0, 1, 2\}$$
$$x_2 \in \{0, 1, 2\}$$
$$x_1 + x_2 \leq 3.5$$

File: `cowspigs.gms`

Variables

Syntax: `[var-type] variable[s] varname [text] {, varname [text]}`

- `var-type` allows to pre-determine the Range of a variable:

Variable type	Range
`free` (default)	$\mathbb{R}$
`positive`	$\mathbb{R}_{\geq 0}$
`negative`	$\mathbb{R}_{\leq 0}$
`binary`	$\{0, 1\}$
`integer`	$\{0, 1, \ldots, 100\}$ (!!)
`semicont`	$\{0\} \cup [\ell, u]$ (default: $\ell = 1$, $u = \infty$)
`semiint`	$\{0\} \cup \{\ell, ell + 1, \ldots, u\}$ (default: $\ell = 1$, $u = 100$)
`sos1`, `sos2`	special ordered sets of type 1 and 2 (later ...)

- Examples:

```
variables x, y, objvar;
positive variable x;
integer  variable z;
```

Variable Attributes

Attributes of a variable:

Attribute	Meaning
`.lo`	lower bound on variable range
`.up`	upper bound on variable range
`.fx`	fixed value for a variable
`.l`	current primal value (updated by solver)
`.m`	current dual value (updated by solver)
`.scale`	scaling factor
`.prior`	branching priority

- Examples:

```
x.up = 10;
y.fx = 5.5;
display z.l;
```

Equations

- `Equations` serve to define restrictions (constraints) and an objective function

Declaration:

- Syntax: `Equation[s] eqnname [text] {, eqnname [text]} ;`
- Example:

```
Equation objective   "Objective Function"
         e1          "First constraint"
```

Equations

- Equations serve to define restrictions (constraints) and an objective function

Declaration:

- Syntax: `Equation[s] eqnname [text] {, eqnname [text]} ;`
- Example:

```
Equation objective   "Objective Function"
         e1          "First constraint"
```

Definition:

- Syntax: `eqnname(domainlist).. expression eqn_type expression;`

eqn_type	meaning
=e=	$=$
=g=	$\geq$
=l=	$\leq$
=n=	no relation between left and right side
=x=	for external functions
=c=	for conic constraints
=b=	for logic constraints

- Example:

```
objective.. objvar =e= 2 * x + 3 * y*y - y + 5 * z ;
e1..        x + y  =l= z;
```

Equation Attributes

Attributes of an equation:

Attribute	Meaning
`.l`	activity (updated by solver)
`.m`	current dual value (updated by solver)
`.scale`	scaling factor

Model statement

- Model = a collection of equations
- Syntax:

```
model[s] model_name [text] [/ all | eqn_name {, eqn_name}]
       {,model_name [text] [/ all | eqn_name {, eqn_name}] };
```

- Example:

```
model m / all /;
model m / objective, e1 /;
```

Model statement

- Model = a collection of equations
- Syntax:

```
model[s] model_name [text] [/ all | eqn_name {, eqn_name}]
       {,model_name [text] [/ all | eqn_name {, eqn_name}] };
```

- Example:

```
model m / all /;
model m / objective, e1 /;
```

Attributes:

set by user		set by solver	
`iterlim`	iteration limit	`iterusd`	number of iterations
`reslim`	time limit in seconds	`resusd`	solving time
`optcr`	relative gap tolerance	`modelstat`	model status
`optfile`	number of solver options file	`solvestat`	solver status
...		...	

Example:

```
m.reslim = 60;      m.optcr = 0;
solve m minimize objvar using MINLP;
display m.resusd;
```

Solving a model

- Passing a model to a solver and evaluation of results
- Specification of one free variable to be minimized or maximized
- Syntax:

```
solve modelname using modeltyp maximizing|minimizing varname|;
solve modelname maximizing|minimizing varname using modeltyp ;
```

- the model type defines the problem class to be used for the model:

`LP`	a linear problem
`QCP`	quadratically constraint problem (only linear and quadratic terms)
`NLP`	a nonlinear problem with continuous functions
`DNLP`	a nonlinear problem with discontinuous functions
`MIP`	a mixed-integer linear problem
`MIQCP`	a mixed-integer quadratically constraint problem
`MINLP`	a mixed-integer nonlinear problem
`CNS`	a nonlinear constraint satisfaction problem (no objective function)
`RMIP`,...	a mixed-integer problem with relaxed integrality restrictions

- Example:

```
solve m using  MINLP minimizing objvar;
solve m using RMINLP min        objvar;
```

GAMS command line call

Calling GAMS from the command line:

`$ gams <modelfile> { [-]key=value | key value }`

Examples:

- `gams trnsport.gms`
- `gams trnsport`
- `gams trnsport.gms LP=CBC`
- `gams trnsport LP CBC`
- `gams trnsport -LP CBC`

Listing File

Running a GAMS model generates a listing file (`.lst` file).

Compilation Errors:

- are indicated by ****
- contain a '$' directly below the point at which the compiler thinks the error occurred
- are explained near the end of the line-numbered listing part
- in the IDE, they are also indicated by red lines in the process (log) window (can be double-clicked)
- check carefully for the cause of the first error, fix it, and try again
- usual causes: undefined / undeclared symbols (parameters, variables, equations), unmatched brackets, missing semi-colons

Listing File: Equation and Column Listing

Equation Listing:

- listing of generated equations with sets unrolled, parameters removed, ...
- useful for model debugging: is the intended model generated?
- for nonlinear equations, a linearization in the starting point is shown

```
   AcidDef..  AcidDilut*AcidErr  =e= 35.82-22.2*F4Perf;
-> AcidDef..  (1)*AcidDilut + 22.2*F4Perf + (3.6)*aciderr =E=
                      35.82 ; (LHS = 35.79, INFES = 0.03 ****)
```

- activity and violation of constraint in starting point also shown

Listing File: Equation and Column Listing

Equation Listing:

- listing of generated equations with sets unrolled, parameters removed, ...
- useful for model debugging: is the intended model generated?
- for nonlinear equations, a linearization in the starting point is shown

```
   AcidDef..  AcidDilut*AcidErr  =e= 35.82-22.2*F4Perf;
-> AcidDef..  (1)*AcidDilut + 22.2*F4Perf + (3.6)*aciderr =E=
                        35.82 ; (LHS = 35.79, INFES = 0.03 ****)
```

- activity and violation of constraint in starting point also shown

Column Listing:

- shows coefficients, bounds, starting values for generated variables

```
-- F4Perf  F4 Performance Number

F4Perf
                (.LO, .L, .UP, .M = 1.45, 1.45, 1.62, 0)
       22.2     AcidDef
       (1)      F4Def
```

Listing File: Solve Summary

- generated for each solve command
- reporting status and result of solve

```
               S O L V E      S U M M A R Y

     MODEL   m                   OBJECTIVE  F
     TYPE    NLP                 DIRECTION  MINIMIZE
     SOLVER  CONOPT              FROM LINE  85

**** SOLVER STATUS     1 Normal Completion
**** MODEL STATUS      2 Locally Optimal
**** OBJECTIVE VALUE                -1.7650

 RESOURCE USAGE, LIMIT          0.006      1000.000
 ITERATION COUNT, LIMIT        16    2000000000
 EVALUATION ERRORS              0             0
```

Listing File: Solution Listing

- equation and variable primal and dual values and bounds
- marking of infeasibilities, "non-optimalities", and unboundedness
- '.' = zero

```
                    LOWER          LEVEL          UPPER         MARGINAL

-- EQU Objective      .              .              .            1.0000
-- EQU AlkylShrnk     .              .              .           -4.6116
-- EQU AcidBal        .           -0.0020           .           11.8406 INFES
-- EQU IsobutBal      .            0.0952           .            0.0563 INFES
-- EQU AlkylDef       .            0.0127           .           -1.0763 INFES
-- EQU OctDef        0.5743        0.5747         0.5743       -25.9326 INFES
-- EQU AcidDef      35.8200       35.8533        35.8200         0.2131 INFES
-- EQU F4Def        -1.3300       -1.3300        -1.3300        -4.1992

                    LOWER          LEVEL          UPPER         MARGINAL

-- VAR F            -INF          -1.4143         +INF             .
-- VAR OlefinFeed     .            1.6198         2.0000        -0.1269  NOPT
-- VAR IsobutRec      .            1.3617         1.6000        -0.2133  NOPT
-- VAR AcidFeed       .            0.7185         1.2000        -0.0411  NOPT
-- VAR AlkylYld       .            2.8790         5.0000        -0.0076  NOPT
-- VAR IsobutMak      .            1.8926         2.0000        -0.4764  NOPT
-- VAR AcidStren     0.8500        0.8998         0.9300         0.5273  NOPT
```

Cows & Pigs Continued

Define:

- a set of animals (`i`)
- a parameter with profit for each animal (`p(i)`)
- a parameter with maximal number for each animal (`xmax(i)`)
- a parameter for the max number of all animals (`maxanimal`)
- an integer variable to count how many of each animal to buy (`x(i)`)
- a real variable to hold the profit (`profit`)
- an equation to define the objective
- an equation to limit the total number of purchased animals by `maxanimal`

Fill sets and parameters with the data from the original example:

- 2 animals: cow and pig
- profit for cow: 3 profit for pig: 2
- maximal number of cows: 2 maximal number of pigs: 2
- maximal number of animals: 3.5

File: `cowspigs2.gms`

Sets

- Basic elements of a model
- Syntax:

```
set set_name ["text"] [/element ["text"] {,element ["text"]} /]
  {,set_name ["text"] [/element ["text"] {,element ["text"]} /]} ;
```

Sets

- Basic elements of a model
- Syntax:

```
set set_name ["text"] [/element ["text"] {,element ["text"]} /]
  {,set_name ["text"] [/element ["text"] {,element ["text"]} /]} ;
```

- Name `set_name` is identifier
- Elements have up to 63 characters, start with letter or digit or are quoted:

```
A    Phos-Acid    September    1986    1952-53    Line-1
'*TOTAL*'    '10%incr'    '12"/foot'    "Line 1"
```

- Elements have no value (!), that is, '1986' does not have the numerical value 1986 and '01' $\neq$ '1'
- Text has up to 254 characters, all in one line

Sets

- Basic elements of a model
- Syntax:

```
set set_name ["text"] [/element ["text"] {,element ["text"]} /]
  {,set_name ["text"] [/element ["text"] {,element ["text"]} /]} ;
```

- Name `set_name` is identifier
- Elements have up to 63 characters, start with letter or digit or are quoted:

```
A    Phos-Acid    September    1986    1952-53    Line-1
'*TOTAL*'    '10%incr'    '12"/foot'    "Line 1"
```

- Elements have no value (!), that is, '1986' does not have the numerical value 1986 and '01' $\neq$ '1'
- Text has up to 254 characters, all in one line
- Example:

```
Set n   Nutrients
/ Prot  "Protein (mg)"
  VitA  "Vitamine A",    VitC   'Vitamine C',
  Calc  Calcium
/;
```

Sequences, Alias

Sequences in Sets: $*$-Notation

- `set t "time" / 2000*2008 /;`
  corresponds to

  `set t "time" / 2000,2001,2002,2003,2004,2005,2006,2007,2008 /;`
- `a1bc*a20bc` is different from `a01bc*a20bc`
- the following are wrong: `a1x1*a9x9`, `a1*b9`

Sequences, Alias

Sequences in Sets: $*$-Notation

- `set t "time" / 2000*2008 /;`
  corresponds to

  `set t "time" / 2000,2001,2002,2003,2004,2005,2006,2007,2008 /;`
- `a1bc*a20bc` is different from `a01bc*a20bc`
- the following are wrong: `a1x1*a9x9`, `a1*b9`

Several names for one set: `alias` command

- Syntax: `alias(set_name, set_name \{, set_name\})`
- Example:

```
set ice / chocolate, strawberry, cherry, vanilla /;
alias(ice, icecreme, gelado, sorvete);
```

Data

- data in GAMS consists always of real numbers (no strings)
- uninitialized data has the default value 0
- 3 forms to declare data:
  - Scalar: a single (scalar) date
  - Parameter: list oriented data
  - Table: table oriented data (at least 2 dimensions)

Data

- data in GAMS consists always of real numbers (no strings)
- uninitialized data has the default value 0
- 3 forms to declare data:
  - Scalar: a single (scalar) date
  - Parameter: list oriented data
  - Table: table oriented data (at least 2 dimensions)

Scalar Data:

- Syntax:

```
scalar[s] scalar_name [text] [/signed_num/]
        { scalar_name [text] [/signed_num/] };
```

- Example:

```
Scalars rho Discountfactor / .15 /
        izf internal rate of return;
```

Data: Parameters

Parameter:

- can be indexed over a one or several sets
- Syntax:

```
parameter[s] param_name [text] [/ element [=] signed_num
                                  {,element [=] signed_num} /]
            {,param_name [text] [/ element [=] signed_num
                                  {,element [=] signed_num} /] } ;
```

- Example:

```
set ice icecreams / chocolate, strawberry, cherry, vanilla /;
parameter demand(ice) / chocolate 50, strawberry = 30
                        vanilla    20 /
```

- Example:

```
set c  'countries'  / jamaica, haiti, guyana, brazil /;
parameter demand(c,ice)  "Demand of icecream per country (t)"
 / Jamaica.Chocolate 300, Jamaica.Strawberry 50, Jamaica.Cherry 5
   Haiti.(Chocolate,Vanilla,Strawberry)  =   30,
   (Guyana,Brazil).Chocolate                100 /
```

Data: Tables

Tables:

- Syntax:

```
table table_name [text] EOL
            element    { element }
    element signed_num { signed_num } EOL
   {element signed_num { signed_num } EOL} ;
```

- Example:

```
table demand(c,ice)  "Demand of icecream per country (t)"
                    Chocolate  Strawberry  Cherry  Vanilla
Jamaica                   300          50       5
Haiti                      30          30                30
(Guyana,Brazilien)        100                              ;
```

- no “free form”: position of elements is of importance
- tables with more than 2 dimensions are also possible

Data: Assignments

- Scalar Assignment:

```
scalar x / 1.5 /;
x = 1.2;
x = x + 2;
```

- Indexed Assignment:

```
Set row          / r1*r10 /
    col          / c1*c10 /
    subrow(row) / r7*r10 /;
Parameter a(row,col), r(row), c(col);
a(row,col)      = 13.2 + r(row)*c(col);
a('r7','c4')    = -2.36;

a(subrow,'c10') = 2.44 - 33*r(subrow);

a(row,row)      = 7.7 - r(row);
alias(row,rowp);
a(row,rowp)     = 7.7 - r(row) + r(rowp);
```

Data: Expressions

Expression: an arbitrarily complex calculation instruction

Arithmetic operators: `**` (exponentiate), +, -, *, /

- ```
  x = 5 + 4*3**2;
  x = 5 + 4*(3**2);
  ```
- `x**n` corresponds to `exp(n*log(x))` $\Rightarrow$ only allowed for $x > 0$
- `population(t) = 56*(1.015**(ord(t)-1))`
  ```

Data: Expressions

Expression: an arbitrarily complex calculation instruction

Arithmetic operators: `**` (exponentiate), +, -, *, /

- ```
  x = 5 + 4*3**2;
  x = 5 + 4*(3**2);
  ```
- `x**n` corresponds to `exp(n*log(x))` ⇒ only allowed for $x > 0$
- `population(t) = 56*(1.015**(ord(t)-1))`

Indexed Operations:

- Syntax: `indexed_op( (controlling_indices), expressions)`
- `indexed_op` can be: `sum`, `prod`, `smin`, `smax`

```
parameter demand(c,ice)  "demand (t)"
          totaldemand(c) "totaler demand per country (t)"
          completedemand "totaler demand for all countries (t)"
          mindemand(ice) "minimal demand per icecream";
totaldemand(c) = sum(ice,     demand(c,ice));
completedemand = sum((ice,c), demand(c,ice));
mindemand(ice) = smin(c,      demand(c,ice));
```
```

Data: Expressions

Expression: an arbitrarily complex calculation instruction

Arithmetic operators: ** (exponentiate), +, -, *, /

- ```
  x = 5 + 4*3**2;
  x = 5 + 4*(3**2);
  ```
- `x**n` corresponds to `exp(n*log(x))` ⇒ only allowed for $x > 0$
- `population(t) = 56*(1.015**(ord(t)-1))` (t-1)

Indexed Operations:

- Syntax: `indexed_op( (controlling_indices), expressions)`
- `indexed_op` can be: sum, prod, smin, smax

```
parameter demand(c,ice)  "demand (t)"
          totaldemand(c) "totaler demand per country (t)"
          completedemand "totaler demand for all countries (t)"
          mindemand(ice) "minimal demand per icecream";
totaldemand(c) = sum(ice,     demand(c,ice));
completedemand = sum((ice,c), demand(c,ice));
mindemand(ice) = smin(c,      demand(c,ice));
```

Functions: `errorf(x)`, `exp(x)`, `power(x,n)`, `sqr(x)`, `uniform(a,b)`, `normal(mean,sdev)`, ...
```

What the Solve Statement is doing

At each Solve statement, the following happens...

1. The model (= list of (indexed) equations) is compiled into a model instance, that is, a scalar (= no indices) list of constraints and those variables, that appear in at least one constraint.
   (The Equation Listing and Column Listing shows in the listing file shows parts of this model.)
2. Instance statistics are written to the log:
   ```
   --- Generating NLP model m
   --- alkyl.gms(85) 5 Mb
   ---   8 rows  15 columns  32 non-zeroes
   ---   54 nl-code  19 nl-non-zeroes
   ```
3. Some error check is performed.
4. The instance is passed on to a solver and processed there.
   ```
   --- Executing IPOPT: elapsed 0:00:00.093
   [solver log]
   --- Restarting execution
   ```
5. The result (model and solution status, solution, statistical information) is passed back to GAMS and reported in the listing file.

Solver Status

Solver status: state of the solver program

1	Normal Completion
2	Iteration Interrupt
3	Resource Interrupt (timelimit reached)
4	Terminated by Solver
5	Evaluation Error Limit
6	Capability Problems
7	Licensing Problems
8	User Interrupt
9, ..., 13	setup, solver, system errors

Model Status

Model status: what the solution looks like

1	Optimal (global optimality)
2, 8	Locally Optimal
3, 18	Unbounded
4, 10, 19	Infeasible (proven to be infeasible)
5	Locally Infeasible
6	Intermediate Infeasible
7	Intermediate Nonoptimal (feasible for continuous models)
9	Intermediate Non-integer
11	License Problem
12, 13	Error
14	No Solution (but expected so)
15, 16, 17	Solved for CNS models

Option command

- specification of systemwide parameters to control output, solving process, ...
- Syntax:
  `option keyword1 [ = value1 ] { ,|EOL keyword2 = [ value2 ] }`
- some important parameters:

keyword	meaning	default
`iterlim`	iteration limit	2000000000
`reslim`	time limit	1000 (!!)
`optca`	absolute gap tolerance	0.0
`optcr`	relative gap tolerance	0.1 (!!)
`LP`	choice of LP solver	CPLEX
`NLP`	choice of NLP solver	CONOPT
...		

- Example:

```
option iterlim = 100, optcr = 0;
solve icesale using mip min kosten;
option mip = cbc;
solve icesale using mip min kosten;
```

- options can also be set on the command line:

```
> gams icesale.gms mip=cbc optcr=0
```

Display command

- display sets, data, variable/equation/model attributes in the listing file
- Examples:

```
display ice, sorbet;
display x.l, x.m;
display demand;
display satdemand.m;
```

- only non-zero values are displayed
- control number of digits after the decimal point for all displayed values

```
option decimals = 1 ;
```

- number of digits after the decimal point for variable `x`:

```
option x:6 ;
```

Outline

Basic Modeling

Dynamic Models

More GAMS

Cows & Pigs by Complete Enumeration (!) in GAMS

x_1	the number of cows to purchase ($x_1 \in \{0, 1, 2\}$)
x_2	the number of pigs to purchase ($x_2 \in \{0, 1, 2\}$)
maximize	$z = 3x_1 + 2x_2$
such that	$x_1 + x_2 \leq 3.5$

Cows & Pigs by Complete Enumeration (!) in GAMS

x_1 the number of cows to purchase ($x_1 \in \{0, 1, 2\}$)

x_2 the number of pigs to purchase ($x_2 \in \{0, 1, 2\}$)

maximize $z = 3x_1 + 2x_2$

such that $x_1 + x_2 \leq 3.5$

```
scalar x1, x2, obj;
scalar objbest, x1best, x2best;
objbest = 0;
for( x1 = 0 to 2,
  for( x2 = 0 to 2,
    if( x1 + x2 le 3,
      obj = 3*x1 + 2*x2;
      if( obj > objbest,
        x1best = x1;
        x2best = x2;
        objbest = obj;
   ) )) )
display x1best, x2best, objbest;
```

File: `cowspigsenum.gms`

Finding a good local optimum to a NLP: Multistart

- Starting an NLP solver from different starting points and pick the best solution.
- If we don't know how to pick a good point, let's pick one randomly.

Finding a good local optimum to a NLP: Multistart

- Starting an NLP solver from different starting points and pick the best solution.
- If we don't know how to pick a good point, let's pick one randomly.
- Multistart algorithm:
    1. $f^U = \infty$
    2. for $k = 1$ to N, do
        2.1 Generate starting point x uniformly at random over $[\underline{x}, \overline{x}]$.
        2.2 Run NLP solver from x and obtain solution x^*.
        2.3 if $f(x^*) < f^U$: $f^U = f(x^*)$ and $x^U = x^*$
    3. Return x^U and f^U.
- The GAMS solver MSNLP implements such an algorithm.

Finding a good local optimum to a NLP: Multistart

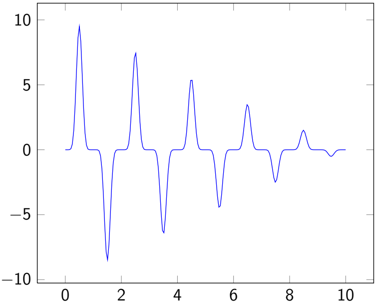
- Starting an NLP solver from different starting points and pick the best solution.
- If we don't know how to pick a good point, let's pick one randomly.
- Multistart algorithm:
  1. $f^U = \infty$
  2. for $k = 1$ to N, do
     - 2.1 Generate starting point x uniformly at random over $[\underline{x}, \overline{x}]$.
     - 2.2 Run NLP solver from x and obtain solution x^*.
     - 2.3 if $f(x^*) < f^U$: $f^U = f(x^*)$ and $x^U = x^*$
  3. Return x^U and f^U.
- The GAMS solver MSNLP implements such an algorithm.

- Example:

$$\max_{x \in [0,10]} \phi(x),$$

$$\text{where } \phi(x) = (10 - x)\sin^9(2\pi x)$$

- Optimal solution:
  $x = 0.24971128$, $\phi(x) = 9.75014433$
- File: `nlpsin.gms`



Program flow control, loop command

- controlling the execution of a GAMS program
- commands: `loop`, `if-else`, `while`, `for`
- declarations and definition of equations are not allowed inside these commands
- solve statements are allowed

Program flow control, loop command

- controlling the execution of a GAMS program
- commands: `loop`, `if-else`, `while`, `for`
- declarations and definition of equations are not allowed inside these commands
- solve statements are allowed

`loop` command:

- Syntax:

```
loop(controllingset[$(condition)],
     statement {; statement}
    );
```

- Example:

```
set t / 1985*1990 /;
parameter pop(t) / 1985  3456 /
          growth(t) / 1985  25.3,  1986  27.3,  1987  26.2,
                      1988  27.1,  1989  26.6,  1990  26.6 /;
loop(t, pop(t+1) = pop(t) + growth(t));
```

If-Elseif-Else command

- Syntax:

```
if( condition, statement {; statement};
{elseif condition, statement {; statement}; }
[else statement {; statement};]
);
```

- Example:

```
if( (ml.modelstat eq 4),
*      model ml was infeasible,
*      relax bound and solve again
  x.up(j) = 2*x.up(j);
  solve ml using lp min z;
elseif ml.modelstat eq 1,
  display x.l;
else
  abort "Error solving the model";
);
```

While, Repeat commands

`While` command:

- Syntax: `while(condition, statement \{; statement\}; );`
- Example:

```
scalar count / 1 /;
scalar globmin / inf /;
while(count le 1000,
  x.l(j) = uniform(x.lo(j), x.up(j));
  solve ml using nlp min obj;
  if ( obj.l le globmin, globmin = obj.l; );
  count = count + 1;
);
```

While, Repeat commands

`While` command:

- Syntax: `while(condition, statement \{; statement\}; );`
- Example:

```
scalar count / 1 /;
scalar globmin / inf /;
while(count le 1000,
  x.l(j) = uniform(x.lo(j), x.up(j));
  solve ml using nlp min obj;
  if ( obj.l le globmin, globmin = obj.l; );
  count = count + 1;
);
```

`Repeat` command:

- Syntax: `repeat(statement \{; statement\}; until condition );`
- Example:

```
scalar count / 1 /;  scalar globmin / inf /;
repeat( x.l(j) = uniform(x.lo(j), x.up(j));
        solve ml using nlp min obj;
        if ( obj.l le globmin, globmin = obj.l; );
        count = count + 1;
until count eq 1000 );
```

For command

```
for(i = start to|downto end [by incr], statement \{; statement\};);
```

- Note: `i` is a scalar, not a set
- `start`, `end`, and `incr` can be real numbers, but `incr` needs to be positive
- Example:

```
scalar count;
scalar globmin / inf /;
for( count = 1 to 1000,
  x.l(j) = uniform(x.lo(j), x.up(j));
  solve ml using nlp min obj;
  if ( obj.l le globmin, globmin = obj.l; );
);
```

Subsets, Cardinality

Subsets:

- Syntax for `set_name1` $\subseteq$ `set_name2`: `set set_name1 (set_name2);`
- Example:

```
set ice / chocolate, strawberry, cherry, vanilla /;
set sorbet(ice) / strawberry, cherry /;
```

- Domain checking: `set sorbet(ice) / strawberry, banana /;` $\Rightarrow$ error

Subsets, Cardinality

Subsets:

- Syntax for set_name1 $\subseteq$ set_name2: `set set_name1 (set_name2);`
- Example:

```
set ice / chocolate, strawberry, cherry, vanilla /;
set sorbet(ice) / strawberry, cherry /;
```

- Domain checking: `set sorbet(ice) / strawberry, banana /;` $\Rightarrow$ error

`Card`(set)

- gives the number of elements in a set
- Example:

```
set c      'countries'  / jamaica, haiti, guyana, brazil /;
scalar nc 'number of countries;
  nc = card(c);
```

Ordered Sets

Lag & Lead Operations:

- allow to access neighbors (next or further distant) of a elements in a priori explicitly specified ordered set
- Syntax: `setelement` $\pm$ n
- Note: `x(setelement+n)` is zero if position of setelement $>$ `card(set)-n`
- Example:

```
Set t / 1*24 /;
Variables level(t), inflow(t), outflow(t);
Equation balance(t)  couple fill levels of reservoir over time;
* implicitly assume an initial fill level of zero
balance(t).. level(t) =e= level(t-1) + inflow(t) - outflow(t);
```

Ordered Sets

Lag & Lead Operations:

- allow to access neighbors (next or further distant) of a elements in a priori explicitly specified ordered set
- Syntax: `setelement` $\pm$ `n`
- Note: `x(setelement+n)` is zero if position of setelement $>$ `card(set)-n`
- Example:

```
Set t / 1*24 /;
Variables level(t), inflow(t), outflow(t);
Equation balance(t)  couple fill levels of reservoir over time;
* implicitly assume an initial fill level of zero
balance(t).. level(t) =e= level(t-1) + inflow(t) - outflow(t);
```

`Ord`(setelement):

- gives position of an element in an a priori explicitly specified ordered sets
- Example:

```
Set t / 1*24 /;
Parameter hour(t);
hour(t) = ord(t);
```

Dynamic Sets

- dynamic sets allow elements to be added or removed
- dynamic sets are usually domain-checked, i.e., subsets
- Syntax: `setname(othersetelement) = yes | no` (add/remove single element)
- Syntax: `setname(subset) = yes | no` (add/remove another subset)
- Example:

```
set ice / chocolate, strawberry, cherry, vanilla /;
Parameter demand(ice) / chocolate 1000, strawberry 500,
                        cherry 10, vanilla 100 /;

set sorbet(ice);
sorbet(ice) = yes;
sorbet('chocolate') = no;   sorbet('vanilla') = no;

Set highdemand(ice)  ice creams with high demand;
highdemand(ice) = (demand(ice) >= 500);
```

- most often used as controlling index in an assignment or equation definition

```
Scalar sumhighdemand;
sumhighdemand = sum(highdemand, demand(highdemand));
```

Example: refer to first/last period of discrete-time models

```
Set t / 1*24 /;
Sets tb(t)  base period
     tn(t)  non-base periods
     tt(t)  terminal period;
tb(t) = (ord(t) = 1);
tn(t) = (ord(t) > 1);
tt(t) = (ord(t) = card(t));
Variables level(t), inflow(t), outflow(t);
Equations balance(t)     couple fill levels of reservoir over time
          basebalance(t) define fill level for base period;
* only for time periods > base period
balance(tn(t)).. level(t) =e= level(t-1) + inflow(t) - outflow(t);
* only for base period
basebalance(tb).. level(tb) =e= 100 + inflow(tb) - outflow(tb);
* lower bound on fill level in terminal period
level.lo(tt) = 100;
```

Alternatively (but less readable):

```
equation basebalance;  basebalance..
   sum(tb, level(tb)) =e= 100 + sum(tb, inflow(tb) - outflow(tb));
```

Outline

Basic Modeling

Dynamic Models

More GAMS

Multidimensional Sets

Multidimensional Sets:

- describing assignments (relations) between sets
- Example:

```
sets c  'countries'  / jamaica, haiti, guyana, brazil /
     h  'harbors'    / kingston, s-domingo, georgetown, belem /
set  hc(h, c)  harbor to country relation
     / kingston.jamaica,   s-domingo.haiti
       georgetown.guyana,  belem.brazil /;
```

sameas and diag

sameas(setelement, otherelement) and sameas(setelement, “text”)
diag(setelement, otherelement) and diag(setelement, “text”)

- sameas returns true if identifiers for given set elements are the same, or if identifier of one set element equals a given text
- sameas can also be used as a set
- diag is like sameas, but return 1 if true, and 0 otherwise
- Example:

```
sets ice1 / chocolate, strawberry, cherry, vanilla /
     ice2 / strawberry, cherry, banana /;
scalar ncommon;
ncommon = sum((ice1, ice2), diag(ice1,ice2));
ncommon = sum(sameas(ice1, ice2), 1);
```

Conditional expressions: $ Operator

Boolean Operators:

- numerical operators: `lt`, `<`, `le`, `<=`, `eq`, `=`, `ne`, `<>`, `ge`, `>=`, `gt`, `>`
- logical operators: `not`, `and`, `or`, `xor`
- set membership: `a(i)` evaluates to *true* if and only if `i` is contained in the (sub)set `a`, otherwise *false*
- *true* corresponds to 1, *false* to 0

Conditional expressions: $ Operator

Boolean Operators:

- numerical operators: `lt`, `<`, `le`, `<=`, `eq`, `=`, `ne`, `<>`, `ge`, `>=`, `gt`, `>`
- logical operators: `not`, `and`, `or`, `xor`
- set membership: `a(i)` evaluates to *true* if and only if `i` is contained in the (sub)set `a`, otherwise *false*
- *true* corresponds to 1, *false* to 0

$-Operator:

- allows to apply necessary conditions
- `$(condition)` can be read as "if `condition` is true"

Cond.

- Example:
  "If `b`>1.5, then let `a`= 2." ⇒ `a$(b > 1.5) = 2;` ✓
  "If `b`>1.5, then let `a`= 2, otherwise let `a`= 0." ⇒ `a = 2$(b > 1.5);`
- $ on left side: no assignment, if `condition` not satisfied
- $ on right side: always assignment, but term with $ evaluates to 0, if `condition` not satisfied
- cannot be used in declarations

Applications for $ Operator

Filtering in indexed operations:

```
parameter sorbetbalance;
set ice; set sorbet(ice);
sorbetbalance = sum(ice$sorbet(ice), price(ice)*purchase.l(ice));
sorbetbalance = sum(sorbet(ice), price(ice) * purchase.l(ice));
```

Applications for $ Operator

Filtering in indexed operations:

```
parameter sorbetbalance;
set ice; set sorbet(ice);
sorbetbalance = sum(ice$sorbet(ice), price(ice)*purchase.l(ice));
sorbetbalance = sum(sorbet(ice), price(ice) * purchase.l(ice));
```

Conditioned indexed operations:

```
rho = sum(i$(sig(i) ne 0), 1/sig(i) - 1);
```

Applications for $ Operator

Filtering in indexed operations:

```
parameter sorbetbalance;
set ice; set sorbet(ice);
sorbetbalance = sum(ice$sorbet(ice), price(ice)*purchase.l(ice));
sorbetbalance = sum(sorbet(ice), price(ice) * purchase.l(ice));
```

Conditioned indexed operations:

```
rho = sum(i$(sig(i) ne 0), 1/sig(i) - 1);
```

Conditioned equations:

```
Equation satdemand(ice);
satdemand(ice)$sorbet(ice).. purchase(ice) =g= demand(ice);

balance(t)$(ord(t)>1)..
                    level(t) =e= level(t-1) + inflow(t) - outflow(t);
```

Existence of variables in constraints:

```
variables x, y;   parameter A;   equation e;
e.. x + y$(A>2) =e= A;
```

No Variables in $ condition

- The following DOES NOT WORK:

```
binary variable x;  variable y;   equation e1, e2;
e1$(x = 1).. y          =l= 100;
e2        .. y$(x = 1) =l= 100;
```

- The value of x is decided by the solver, not by the GAMS.
- However, GAMS has to evaluate $-operators when assembling an instance in the Solve statement.
- Instead, you have to reformulate:

```
e.. y =l= 100 * x + y.up * (1-x);
```

That is: x=1 $\Rightarrow$ y$\leq$ 100; x=0 $\Rightarrow$ y$\leq$y.up.

Compilation vs. Execution

GAMS processed models in 2 phases:

Compilation Phase:

- reads the complete GAMS model and translate into GAMS specific byte code
- processes declarations (variables, equations, sets, parameters)
- execute all compile-time commands (next slides)
- Listing file:

```
GAMS 24.1.2  r40979 Released Jun 16, 2013 LEX-LEG x86_64/Linux

3 16:03:38 Page 1
A Transportation Problem (TRNSPORT,SEQ=1)
C o m p i l a t i o n

<echo of all processed lines>

COMPILATION TIME     =        0.002 SECONDS      3 MB  24.1.2 r4
...
```

Execution Phase:

- executes commands in program: assignments, solve, loop, while, for, if, ...

Compile Time Commands

- introduced by $ sign in the first column (!)
- Syntax: `$commandname argumentlist {commandname argumentlist}`
- modify behavior of GAMS compiler, e.g., how to process the input, customizing output
- allow program flow control on compilation level: call external programs, include other files, if-else, goto, ...
- modifying and reading compile-time variables

Examples:

- define a title for your GAMS program

```
$Title  A Transportation Problem
```

- define a section that contains only comments

```
$onText
This problem finds a least cost shipping schedule that meets
requirements at markets and supplies at factories.
$offText
```

- disable echoing of input lines in listing file

```
$offListing
```

Compile Time Variables

- compile-time variables hold strings
- their value is accessed to via the `%variablename%` notation
- values are assigned via `$set` or `$eval` commands or on the GAMS command line via double-dash-options: `gams --variablename variablevalue`
- for `$eval`, the variable value string is interpreted as numerical expression
- Example:

```
$set  N    10
$eval Nsqr %N% * %N%
set i / 1 * %N% /;
```

- variants `$setLocal`, `$setGlobal`, `$evalLocal`, `$evalGlobal` allow to control the (file)scope of a variable

Compile Time Program Control: $If

- `$If` allows to do conditional processing
- Syntax:
  `$If [not] (exist filename | string1 == string2) new_input_line`
- only one-line clauses allowed (`new_input_line` can be on next line, though)
- Examples:

```
$if exist myfile.dat $log "myfile.dat exists, yeah!"

scalar a;
$if     %difficulty% == easy a = 5;
$if not %difficulty% == easy a = 10;
```

Compile Time Program Control: $If

- `$If` allows to do conditional processing
- Syntax:
  `$If [not] (exist filename | string1 == string2) new_input_line`
- only one-line clauses allowed (`new_input_line` can be on next line, though)
- Examples:

```
$if exist myfile.dat $log "myfile.dat exists, yeah!"

scalar a;
$if     %difficulty% == easy a = 5;
$if not %difficulty% == easy a = 10;
```

- `$IfThen-$ElseIf-$Else-$Endif` allows to control activity for a set of statements; Example:

```
scalar a;
$ifthen %difficulty% == easy
a = 5;
$else
a = 10;
$endif
```

Compile Time Program Control: $Goto

- `$Goto-$Label` allows to skip over or repeat sections of the input
- Example:

```
scalar a / 5 /;
$if %difficulty% == easy $goto easy
a = 10;
$label easy
```

Executing Shell Commands

- $Call passes a following string to the current shell and waits for the command to be completed
- if the string starts with a '=', the operating system is called directly, i.e., no shell is invoked
- Example:

```
$call "gamslib trnsport"
$call "=gams trnsport"

$if exist myfile.dat $call cp myfile.dat mycopy.dat
```

- the errorLevel functions allows to check whether a previous command (e.g., $call) executed without error:

```
$call "gamslib trnsport"
$call "gams trnsport"
$if errorlevel 1 $abort "ouch! - solving trnsport failed"
```

Writing text files

- $Echo and $onEcho-$offEcho allows to write to text files
- Example:

```
$Echo "hello, world!"  > myfile.txt
$OnEcho >> myfile.txt
ahoy-hoy!
$OffEcho
```

- > myfile.txt creates a new file myfile.txt, thereby overwriting a possibly existing one of the same name
- >> myfile.txt appends to an existing file myfile.txt
- Recall: These are compilation-time commands! Not usable to write solve outcomes or similar; use display command or put-facility (later) for this.

Including text files

- `$Include` allows to include ASCII files into a GAMS program
- compilation is then continued for the included file
- Example:

```
Parameter d(i,j) distance in thousands of miles;
$include dist.inc
```

  where `dist.inc` contains

```
Table d(i,j)  distance in thousands of miles
                 new-york       chicago      topeka
    seattle         2.5           1.7          1.8
    san-diego       2.5           1.8          1.4   ;
```

Including csv files

- $OnDelim enables comma separated value (csv) format for data in table or parameter statements

Examples:

```
Table d(i,j)  distance
$ondelim
$include dist.csv
$offdelim
;
```

where dist.csv is

```
,new-york,chicago,topeka
seattle,2.5,1.7,1.8
san-diego,2.5,1.8,1.4
```

```
Parameter d(i,j)  distance /
$ondelim
$include dist.txt
$offdelim
/;
```

where dist.txt is

```
SEATTLE,NEW-YORK,2.5
SAN-DIEGO,NEW-YORK,2.5
SEATTLE,CHICAGO,1.7
SAN-DIEGO,CHICAGO,1.8
SEATTLE,TOPEKA,1.8
SAN-DIEGO,TOPEKA,1.4
```

Put Command

- writing text files at execution time
- associating an identifier `fileid` with a file: `file fileid / myfile.txt /;`
- select a stream (and thus a file) to write to: `put fileid;`
- writing some items (text, labels, numbers): `put item {, item};`
- write a linebreak: `put /;`
- close a stream: `putclose;`

Put Command

- writing text files at execution time
- associating an identifier `fileid` with a file: `file fileid / myfile.txt /;`
- select a stream (and thus a file) to write to: `put fileid;`
- writing some items (text, labels, numbers): `put item {, item};`
- write a linebreak: `put /;`
- close a stream: `putclose;`
- Example:

```
file fx /result.txt/;
put fx 'Shipped quantities between plants and markets' /;
put    '---------------------------------------------' /;
loop((i,j)$x.l(i,j),
  put 'Shipment from ', i.te(i):10, ' to ', j.te(j):10,
      ' in cases:', x.l(i,j) /;
); putclose;
```

gives

```
Shipped quantities between plants and markets
---------------------------------------------
Shipment from seattle    to new-york  in cases:   50.00
Shipment from seattle    to chicago   in cases:  300.00
Shipment from san-diego  to new-york  in cases:  275.00
Shipment from san-diego  to topeka    in cases:  275.00
```

Text Items, Formatted Output

Label names and explanatory texts can be accessed via attributes:

- `ident.ts`: text associated with identifier
- `element.tl`: label associated with set element
- `set.te(element)`: text associated with element of set

Text Items, Formatted Output

Label names and explanatory texts can be accessed via attributes:

- `ident.ts`: text associated with identifier
- `element.tl`: label associated with set element
- `set.te(element)`: text associated with element of set

Local Item Formatting:

- Syntax for formatting item output: `item:{<>}width:decimals`
- `{<>}` specifies whether justified left (<), right (>), or centered (<>)
- `width` is the field width
- `decimals` is the number of decimals for numeric output
- each can be omitted, e.g., `x.l::5`

Text Items, Formatted Output

Label names and explanatory texts can be accessed via attributes:

- `ident.ts`: text associated with identifier
- `element.tl`: label associated with set element
- `set.te(element)`: text associated with element of set

Local Item Formatting:

- Syntax for formatting item output: `item:{<>}width:decimals`
- `{<>}` specifies whether justified left (<), right (>), or centered (<>)
- `width` is the field width
- `decimals` is the number of decimals for numeric output
- each can be omitted, e.g., `x.l::5`

Global Item Formatting:

- change field justification and width for all items of a type
- `.lj`, `.nj`, `.sj`, `.tj`, `.lw`, `.nw`, `.sw`, `.tw` attributes of stream identifier
- see GAMS User's Guide Section 15.10

Text Items, Formatted Output

Label names and explanatory texts can be accessed via attributes:

- `ident.ts`: text associated with identifier
- `element.tl`: label associated with set element
- `set.te(element)`: text associated with element of set

Local Item Formatting:

- Syntax for formatting item output: `item:{<>}width:decimals`
- `{<>}` specifies whether justified left (<), right (>), or centered (<>)
- `width` is the field width
- `decimals` is the number of decimals for numeric output
- each can be omitted, e.g., `x.l::5`

Global Item Formatting:

- change field justification and width for all items of a type
- `.lj`, `.nj`, `.sj`, `.tj`, `.lw`, `.nw`, `.sw`, `.tw` attributes of stream identifier
- see GAMS User's Guide Section 15.10

Cursor Positioning:

- `put @n;` moves cursor to column `n` of current line