GAMS AND PYOMO CHEAT SHEET

GAMS INSTALLATION PROCEDURE:

1. Visit https://www.gams.com/download/ for the latest release, and to obtain the demo license. This page also contains links to older versions of GAMS.

PYOMO INSTALLATION PROCEDURE:

In this course, we run Pyomo on Google Colaboratory (https://colab.research.google.com/notebooks/intro.ipynb)

1. Create a Google account to use Google Colab.
2. Install Pyomo using the following commands,

```
!pip install pyomo
import pyomo.environ as pyomo
```

The Pyomo installation is valid as long as the runtime is connected. If the session is interrupted, or runtime expires, run the installation commands again before running the model code blocks.

GAMS	PYOMO

1. A) **Set definition (elementwise)**
 	
E.g., C = {'car1', 'car2', 'car3'}
 	
`Set C /car1, car2, car3/;`
 	
B) **Set definition (range)**
 	
E.g., D = {1,2,3,4,5,6}
 	
`Set D /1*6/;`
 	
C) **Subset definition**
 	
E.g., E = {'car1', 'car2', 'car3', 'car4', 'car5'}, $F \subseteq E$ = {'car1', 'car4'}
 	
`Set E /car1, car2, car3, car4, car5/;`
 	
`Set F(E);`
`F('car1') = 'yes';`
`F('car4') = 'yes';`
	1. A) **Set definition (elementwise)**
	
	E.g., C = {'car1', 'car2', 'car3'}
	
	`model_name.C = pyomo.Set(initialize = ['car1', 'car2', 'car3'];`
	
	B) **Set definition (range)**
	
	E.g., D = {1,2,3,4,5,6}
	
	`model_name.D = pyomo.RangeSet(6);`
	
	C) **Subset definition**
	
	E.g., E = {'car1', 'car2', 'car3', 'car4', 'car5'}, $F \subseteq E$ = {'car1', 'car4'}
	
	`model_name.E = pyomo.Set(initialize = ['car1', 'car2', 'car3', 'car4', 'car5']);`
	`model_name.F = pyomo.Set(within = model_name.E, initialize = ['car1', 'car4']);`

2. A) **Variable definition**

E.g., x defined over set $C \rightarrow x_C$

```
Variable x(C);
```

B) **Variable domain**

E.g., $x_C \in \mathbb{R}^+$

```
Positive Variable x(C);
```

C) **Variable bounds**

E.g., $x_C \geq 10, x_C \leq 500$

```
x.lo(C) = 10;
x.up(C) = 500;
```

D) **Variable initialization (constant value)**

E.g., Starting solution for $x_C = 11$

```
x.l(C) = 11;
```

E) **Variable initialization (function)**

E.g., Starting solution for $x_C = 10C$

Here, the GAMS function **ord**(set_name) denotes the relative position of an element in the set

```
x.l(C) = 10*ord(C);
```

2. A) **Variable definition**

E.g., x defined over set $C \rightarrow x_C$

```
model_name.x = pyomo.Var(model_name.C);
```

B) **Variable domain**

E.g., $x_C \in \mathbb{R}^+$

```
model_name.x = pyomo.Var(model_name.C,
                         domain = pyomo.PositiveReals);
```

C) **Variable bounds**

E.g., $x_C \geq 10, x_C \leq 500$

```
model_name.x = pyomo.Var(model_name.C, domain =
                         pyomo.PositiveReals, bounds = (10,500));
```

D) **Variable initialization (constant value)**

E.g., Starting solution for $x_C = 11$

```
model_name.x = pyomo.Var(model_name.C, domain =
                   pyomo.PositiveReals, bounds = (10,500), initialize = 11);
```

E) **Variable initialization (function)**

E.g., Starting solution for $x_C = 10C$

Here, we define a function in Python using the **def** command with function name ***init***, function arguments ***model*** and set element *k*

```
def init(model_name,k):
    return 10*k

model_name.x = pyomo.Var(model_name.C, domain =
                   pyomo.PositiveReals, bounds = (10,500), initialize = init);
```

3. A) **One-Dimensional Parameter**

E.g., Parameter *alpha* defined over set C = {'car1', 'car2', 'car3'} → α_C

```
Parameter alpha(C)
/car1   4
 car2   6
 car3   5/;
```

B) **Multi-Dimensional Parameter**

E.g., Parameter *beta* defined over sets:
C = {'car1', 'car2', 'car3'},
L = {'option1', 'option2', 'option3'}
P = {'red', 'black', 'grey'} → $\beta_{C,L,P}$

```
Parameter beta(C, L, P)
/car1.option1.red       13
 car1.option1.black     15
 car1.option1.grey      14
 car1.option2.red       20
 car1.option2.black     23

car1.option2.grey       22
car1.option3.red        27
car1.option3.black      31
car1.option3.grey       29
car2.option1.red        9
car2.option1.black      12
car2.option1.grey       13
car2.option2.red        17
car2.option2.black      21
car2.option2.grey       19
car2.option3.red        22
car2.option3.black      25
car2.option3.grey       23
car3.option1.red        21
car3.option1.black      20
car3.option1.grey       20
car3.option2.red        25
car3.option2.black      27
car3.option2.grey       24
car3.option3.red        31
car3.option3.black      29
car3.option3.grey       28/;
```

3. A) **One-Dimensional Parameter**

E.g., Parameter *alpha* defined over set C = {'car1', 'car2', 'car3'} → α_C

```
model_name.alpha = pyomo.Param(model_name.C,
                               initialize = {'car1':4,'car2':6, 'car3':5});
```

B) **Multi-Dimensional Parameter**

E.g., Parameter *beta* defined over sets:
C = {'car1', 'car2', 'car3'},
L = {'option1', 'option2', 'option3'}
P = {'red', 'black', 'grey'} → $\beta_{C,L,P}$

```
model_name.beta = pyomo.Param(model_name.C, initialize =
        {('car1', 'option1', 'red': 13), ('car1', 'option1', 'black': 15),
         ('car1', 'option1', 'grey': 14), ('car1', 'option2', 'red': 20),
         ('car1', 'option2', 'black': 23), ('car1', 'option2', 'grey': 22),
         ('car1', 'option3', 'red': 27), ('car1', 'option3', 'black': 31),

         ('car1', 'option3', 'grey': 29), ('car2', 'option1', 'red': 9),
        ('car2', 'option1', 'black': 12), ('car2', 'option1', 'grey': 13),
        ('car2', 'option2', 'red': 17), ('car2', 'option2', 'black': 21),
        ('car2', 'option2', 'grey': 19), ('car2', 'option3', 'red': 22),
        ('car2', 'option3', 'black': 25), ('car2', 'option3', 'grey': 23),
        ('car3', 'option1', 'red':  21), ('car3', 'option1', 'black': 20),
        ('car3', 'option1', 'grey': 20), ('car3', 'option2', 'red': 25),
        ('car3', 'option2', 'black': 27), ('car3', 'option2', 'grey': 24),
        ('car3', 'option3', 'red': 31), ('car3', 'option3', 'black': 29),
        ('car3', 'option3', 'grey': 28)});
```

C) **Two-Dimensional Table**

E.g., Parameter *theta* defined over sets:
C = {'car1', 'car2', 'car3'}
W = {'warehouse1', 'warehouse2'} $\rightarrow \theta_{C,W}$

```
Table theta(C)
        warehouse1    warehouse2
car1    4             5
car2    6             3
car3    5             4
;
```

D) **Multi-Dimensional Table**

E.g., Parameter *gamma* defined over sets:
C = {'car1', 'car2', 'car3'}
W = {'warehouse1', 'warehouse2'}
S = {'city1', 'city2'} $\rightarrow \gamma_{C,W,S}$

```
Table gamma(C)
                    city1    city2
car1.warehouse1     3        7
car1.warehouse2     5        4
car2.warehouse1     4        6

car2.warehouse2     9        8
car3.warehouse1     13       10
car3.warehouse2     12       11
;
```

4. A) **Constraint (non-indexed)**

E.g., $\sum_C x_C = 100$

```
Equation eq1 'Eq 1 description';
eq1..   sum(C, x(C)) =e= 1;
```

B) **Constraint (un-indexed)**

E.g., $x_C + y_C \geq \alpha_C, \forall c \in C$

```
Equation eq2(C) 'Eq 2 description';
eq2(C)..   x(C) + y(C) =g= alpha(C);
```

4. A) **Constraint (non-indexed)**

E.g., $\sum_C x_C = 100$

```
def rule1(model_name):
  return sum(model_name.x[C] for C in model_name.C) == 1
model_name.eq1 = pyomo.Constraint(rule = rule1,
                                  doc = 'Eq1 description');
```

B) **Constraint (un-indexed)**

E.g., $x_C + y_C \geq \alpha_C, \forall c \in C$

```
def rule2(model_name):
  return model_name.x[C] + model_name.y[C] >=model_name.alpha[C]
```

	model_name.eq1 = pyomo.Constraint(model_name.C, rule = rule2,bdoc = 'Eq2 description');

5. **Objective function**
 	
E.g., min $z = \sum_C \sum_W \theta_{C,W} x_C$
 	
Equation obj 'Objective function';
obj.. z =e=
sum(C,sum(W,theta(C,W)*x(C)));
	5. **Objective function**
	
	E.g., min $z = \sum_C \sum_W \theta_{C,W} x_C$
	
	def obj_rule(model_name):
	return sum(sum(model_name.theta[C,W]* model_name.x[C]
	for W in model_name.W) for C in model_name.C)
	model_name.eq1 = pyomo.Objective(rule = obj_rule,
	sense = pyomo.minimize);

6. A) **Model definition (all constraints)** Model model_name /all/; B) **Model definition (specific constraints)** Model model_name /obj,eq1,eq2/; In GAMS, model definition is performed after all model components (sets, variables, parameters, bounds, initialization, constraints, and objective function) have been defined.	6. **Model definition** model_name = pyomo.ConcreteModel(); In Pyomo, model definition is performed at the beginning of the code block, and all model components (sets, variables, parameters, bounds, initialization, constraints, and objective function) are defined after this, specific to the model name provided. The solve command is specific to the model name.
7. **Solve command** Solve model_name as ***model_type*** minimizing z; In GAMS, the different model types available are LP, NLP, MIP, MINLP, QCP, MIQCP, DNLP, RMIP, RMINLP, MCP, CNS, MPEC, RMPEC, EMP, and MPSGE. Each model type has a pre-set default solver, set by GAMS. To specify a certain solver, the following command can be added to the code before the solve command,	7. **Solve command** results = pyomo.SolverFactory('solver_name', executable = 'executable_location').solve(model_name); For a complete list of solvers for each model type, visit the Pyomo documentation page. For the location of the executable file for each solver in Google Collaboratory, visit the Google Colab documentation page. If you are running Pyomo locally on your computer, provide the appropriate executable file path. To add solvers in Pyomo on Google Colab, the following installation commands need to be run before the code block,

option ***model_type*** = ***solver_name***;

For a complete list of solvers for each model type, visit the GAMS documentation page. The use of solvers is also limited by the GAMS license held.

If the objective function is to be maximized, the following solve command can be used,

Solve model_name as ***model_type*** maximizing z;

For the solver 'GLPK':

```
!apt-get install -y -qq glpk-utils
```

The 'executable_location' for GLPK in Google Colab is: '/usr/bin/glpsol'

For the solver 'IPOPT':

```
!wget -N -q "https://ampl.com/dl/open/ipopt/ipopt-linux64.zip"
!unzip -o -q ipopt-linux64
```

The 'executable_location' for IPOPT in Google Colab is: '/content/ipopt'

8. **NEOS server usage**

A) **Submitting jobs through webpage**

Visit https://neos-server.org/neos/ → *Submit a job to NEOS* → Browse through *Problem Type* or *Solver* and choose solver → Choose [GAMS] → Use the *Web Submission Form* to submit model.

B) **Direct interface (32 Distribution and newer)**

Solve any GAMS model with chosen solver

(also supported on NEOS) by navigating to the *GAMS* menu on the toolbar → *Run NEOS*.

8. **NEOS server usage**

Before running the model code block, add the package 'os' as,

```
import os;
```

In the model code block, after defining model and its components, replace the solve command defined in (7) with,

```
os.environ['NEOS_EMAIL'] = 'email_address';
solver_manager = pyomo.SolverManagerFactory('neos');
```

```
results = solver_manager.solve(model,opt=solver_name);
```

The solvers available on NEOS for use through Pyomo are: 'bonmin', 'cbc', 'conopt', 'couenne', 'cplex', 'filmint', 'filter', 'ipopt', 'knitro', 'l-bfgs-b', 'lancelot', 'lgo', 'loqo', 'minlp', 'minos', 'minto', 'mosek', 'ooqp', 'path', 'raposa', 'snopt'

9. **Data import**

A) **From text file**

E.g., Parameter *alpha* defined over set C = {'car1', 'car2', 'car3'} → α_C

```
Parameter alpha(C);
$include alpha_data.txt
```

where alpha_data.txt contains the lines,

9. **Data import**

A) **From json file**

E.g., Parameter *alpha* defined over set C = {'car1', 'car2', 'car3'} → α_C

```
data = pyomo.DataPortal();
data.load(filename=alpha_data.json');

model.alpha = pyomo.Param(model.C, initialize = data['alpha']);
```

```
alpha('car1') = 4;
alpha('car2') = 6;
alpha('car3') = 5;
```

B) **From (.xls) spreadsheet file**

```
$call gdxxrw i=beta_data.xls
o=beta_output.gdx par=beta
rng=sheet1!A2:E11 rDim=2 cDim=1
trace=int

$gdxin beta_output.gdx
$load beta
$gdxin
```

where beta_data.xls contains the data,

	A	B	C	D	E
1	**Parameter *beta***				
2			red	black	grey
3	car1	option1	13	15	14
4	car1	option2	20	23	22
5	car1	option3	27	31	29
6	car2	option1	9	12	13
7	car2	option2	17	21	19
8	car2	option3	22	25	23
9	car3	option1	21	20	20
10	car3	option2	25	27	24
11	car3	option3	31	29	28

The 'trace' command signifies extent of detail of the import command in the log,

where *int* takes values between 0 and 4; higher the value of *int*, more the level of import detail in the log.

where alpha_data.json contains the lines,

```
{"alpha":{"car1":4,"car2":6,"car3":5}}
```

B) **From (.xls) spreadsheet file**

```
beta = pd.read_excel('beta_output.xls',
                     sheet_name='Sheet1',header=1,
                     index_col=[0,1],usecols='A:E',nrows=9).fillna('');
```

where beta_data.xls contains the data,

	A	B	C	D	E
1	**Parameter *beta***				
2			red	black	grey
3	car1	option1	13	15	14
4	car1	option2	20	23	22
5	car1	option3	27	31	29
6	car2	option1	9	12	13
7	car2	option2	17	21	19
8	car2	option3	22	25	23
9	car3	option1	21	20	20
10	car3	option2	25	27	24
11	car3	option3	31	29	28

To use the 'pd.read_excel' command, the following package needs to be imported before this code block is run,

```
import pandas as pd
```

The option 'header=1' denotes that the first row of the specified cell range contains data labels (here, 'red', 'black', and 'grey').

The option 'index_col = [0,1]' denotes that the first and second columns of the specified cell range contain data labels (here, 'car1', 'car2', 'car3', and 'option1', 'option2', and 'option3').

The option 'usecols = A:E' specifies the column range containing the data to be imported.

The option 'nrows=9' specifies the number of rows containing the data to be imported.

The suffix fillna('') ensures that any blank cells in the spreadsheet are not automatically imported as 'NA' values (not available).

10. **Data export**

A) **To text file**

```
File file_name
/ file_name.txt/;
file_name.pc = 5;

put file_name;
put "Model status: " model_name.modelstat
/;
put "Solver status: " model_name.solvestat
/;
put "Objective: " z.l /;
put "Variable x: " /;
loop (C,
     put C.tl x.l(C) /
);
putclose;
```

The 'put' writing facility in GAMS can be used to export solutions to a text file named 'file_name'.

The suffix '.l' denotes the level value, which is the optimal variable value reported by the solver.

The model status can be written to the text file using the identifier '*model_name*.modelstat'.

The solver status can be written to the text file using the identifier '*model_name*.solvestat'.

An unindexed variable, such as the objective function value denoted here by z, can be written using its level value (z.l).

A 'loop' can be used to iteratively write the values of the variable x(C) for each index C.

B) **To gdx file**

```
execute_unload 'file_name.gdx', C, z, x;
```

10. **Data export**

A) **To text file**

i) **Using implicit command**

```
model_name.solutions.store_to(results);
results.write(filename = 'file_name.txt')
```

The implicit Pyomo command '*model_name*.solutions.store_to' can be used to write the results to a text file named '*file_name*.txt'.

ii) **Using explicit command**

```
f = open('file_name.txt','w');
f.write("\nObjective function value = %d" % model.obj());
for C in model.C:
  f.write("\nVariable value for index %s = %d" %
  (C,model.x[C]()));
f.close()
```

The 'write' facility in Python can be used to write the results to a text file named '*file_name*.txt' explicitly. Indexed variables can be written to the output file using a 'for' loop.

B) **To .json file**

i) **Using implicit command**

```
model_name.solutions.store_to(results);
```

The 'execute_unload' command can be used to a gdx file containing the set C, objective function value z, and variable x(C).

C) **To (.xlsx) spreadsheet file**

```
execute_unload "file_name.gdx" x.l z;

execute 'gdxxrw.exe file_name.gdx o=
results.xlsx var=x.l rng=Sheet1!B1';

execute 'gdxxrw.exe file_name.gdx o=
results.xlsx var=z rng=Sheet1!E1 ';
```

To import the solution from GAMS to a spreadsheet file (.xlsx format), we first export it to a gdx format.

We follow this by writing the results from the created gdx file, to a spreadsheet, by specifying the variable to be written ('var'), and the location to which the data is to be exported, specified by 'rng'.

```
results.write(filename = 'file_name.json')
```

The implicit Pyomo command '*model_name*.solutions.store_to' can be used to write the results to a json file named '*file_name*.json'.

ii) **Using explicit command**

```
f = open('file_name.json','w');
f.write("\nObjective function value = %d" % model.obj());
for C in model.C:
   f.write("\nVariable value for index %s = %d" %
   (C,model.x[C]()));
f.close()
```

The 'write' facility in Python can be used to write the results to a .json file named '*file_name*.json' explicitly. Indexed variables can be written to the output file using a 'for' loop.

C) **To (.xlsx) spreadsheet file**

i) **Using implicit command**

```
model_name.solutions.store_to(results);
results.write(filename = 'file_name.xlsx')
```

The implicit Pyomo command '*model_name*.solutions.store_to' can be used to write the results to a spreadsheet file named '*file_name*.xlsx'.

ii) **Using explicit command**

```
wb = xlsxwriter.Workbook("file_name.xlsx");

sheet = wb.add_worksheet();


row_headers_x = model.C.data();

for item in range(len(row_headers_x)):
   sheet.write(item,0,"%s" % model.C[item+1]

for c in range(len(row_headers_x)):
   cc = model.C[c+1];
   sheet.write(cc,1,"%d" % model.x[cc].value);

sheet.write("A5","Objective function value = ");
sheet.write("B5",model.obj());

wb.close();
```

	Note: In Python, the ‘range’ count begins from 0. The set indices in Pyomo begin from 1. Hence, ‘+1’ is added. To use the ‘xlsxwriter.Workbook’ command and all associated functionalities, the following package needs to be imported before this code block is run, !pip install xlsxwriter import xlsxwriter