sasoptpy Documentation

Release 0.1.2

SAS Institute Inc.

Apr 24, 2018

CONTENTS

PDF Version

Date: Apr 24, 2018 **Version**: 0.1.2

sasoptpy is a Python package providing a modeling interface for SAS Viya Optimization solvers. It provides a quick way for users to deploy optimization models and solve them using SAS Viya Optimization Action Set.

sasoptpy currently can handle linear optimization and mixed integer linear optimization problems. Users can benefit from native Python structures like dictionaries, tuples, and list to define an optimization problem. **sasoptpy** uses Pandas structures extensively.

Under the hood, **sasopty** uses swat package to communicate SAS Viya, and uses saspy package to communicate SAS 9.4 installations.

sasoptpy is an interface to SAS Optimization solvers. Check SAS/OR and PROC OPTMODEL for more details about optimization tools provided by SAS and an interface to model optimization problems inside SAS.

See our SAS Global Forum paper: Optimization Modeling with Python and SAS Viya

CHAPTER

ONE

WHAT'S NEW

This page outlines changes from each release.

1.1 v0.1.2 (April 24, 2018)

1.1.1 New Features

- As an experimental feature, **sasoptpy** supports *saspy* connections now
- `Model.solve_local()` method is added for solving optimization problems using SAS 9.4 installations
- `Model.drop_variable()`, `Model.drop_variables()`, `Model.drop_constraint()`, `Model.drop_constraints()` methods are added
- `Model.get_constraint()` and `Model.get_constraints()` methods are added to grab `Constraint` objects in a model
- `Model.get_variables()` method is added
- `_dual` attribute is added to the `Expression` objects
- `Variable.get_dual()` and `Constraint.get_dual()` methods are added
- `Expression.set_name()` method is added

1.1.2 Changes

- Session argument accepts `saspy.SASsession` objects
- `VariableGroup.mult()` method now supports `pandas.DataFrame`
- Type check for the `Model.set_session()` is removed to support new session types
- Problem and solution summaries are not being printed by default anymore, see `Model.get_problem_summary()` and `Model.get_solution_summary()`
- The default behavior of dropping the table after each solve is changed, but can be controlled with the `drop` argument of the `Model.solve()` method

1.1.3 Bug Fixes

- Fixed: Variables do not appear in MPS files if they are not used in the model
- Fixed: `Model.solve()` primalin argument does not pass into options

1.1.4 Notes

- A .gitignore file is added to the repository.
- A new example is added: Decentralization.
- Both *CAS/Viya* and *SAS* versions of the new example are available.
- There is a known issue with the nondeterministic behavior when creating MPS tables. This will be fixed with a hotfix after the release.
- A new option (no-ex) is added to makedocs script for skipping examples when building docs.

1.2 v0.1.1 (February 26, 2018)

1.2.1 New Features

- Initial value argument 'init' is added for `Variable` objects
- `Variable.set_init()` method is added for variables
- Initial value option 'primalin' is added to `Model.solve()` method
- Table name argument 'name', table drop option 'drop' and replace option 'replace' are added to `Model.solve()` method
- Decomposition block implementation is rewritten, block numbers does not need to be consecutive and ordered `Model.upload_user_blocks()`
- `VariableGroup.get_name()` and `ConstraintGroup.get_name()` methods are added
- `Model.test_session()` method is added for checking if session is defined for models
- `quick_sum()` function is added for faster summation of `Expression` objects

1.2.2 Changes

- methods.py is renamed to utils.py

1.2.3 Bug Fixes

- Fixed: Crash in VG and CG when a key not in the list is called
- Fixed: get_value of pandas is depreceated
- Fixed: Variables can be set as temporary expressions
- Fixed: Ordering in `get_solution_table()` is incorrect for multiple entries

1.3 v0.1.0 (December 22, 2017)

- Initial release

CHAPTER

TWO

INSTALLATION

2.1 Python version support and dependencies

sasoptpy is developed and tested for Python version 3.5+.

It depends on the following packages:

- numpy
- saspy (Optional)
- swat
- pandas

2.2 Getting swat

swat should be available to use SAS Viya solvers.

swat releases are listed at https://github.com/sassoftware/python-swat/releases. After downloading the platform-specific release file, it can be installed using pip:

```
pip install python-swat-X.X.X-platform.tar.gz
```

2.3 Getting saspy

saspy should be available to use SAS 9.4 solvers. The **sasoptpy** support for SAS 9.4 solvers is experimental. Note that saspy is not a requirement for using the SAS Viya solvers.

saspy releases are listed at https://github.com/sassoftware/saspy/releases. The easiest way to download the latest stable version of saspy is to use:

```
pip install saspy
```

2.4 Getting sasoptpy

The latest release of **sasoptpy** can be obtained from the online repository. Call:

```
git clone https://github.com/sassoftware/sasoptpy.git
```

Then inside the sasoptpy folder, call:

```
pip install .
```

Alternatively, you can use:

```
python setup.py install
```

2.5 Step-by-step installation

1. **Installing pandas and numpy**

   First, download and install numpy and pandas using pip:

   ```
   pip install numpy
   pip install pandas
   ```

2. **Installing the swat package**

   First, check the swat release page to find the latest release of the SAS-SWAT package for your environment.

   Then install it using

   ```
   pip install python-swat-X.X.X.platform.tar.gz
   ```

   As an example, run

   ```
   wget https://github.com/sassoftware/python-swat/releases/download/v1.2.1/python-
   ↪swat-1.2.1-linux64.tar.gz
   pip install python-swat-1.2.1-linux64.tar.gz
   ```

   to install the version 1.2.1 of the swat package for 64-bit Linux environments.

3. **Installing sasoptpy**

   Finally you can install *sasoptpy* by downloading the latest archive file and install via pip.

   ```
   wget *url-to-sasoptpy.tar.gz*
   pip install sasoptpy.tar.gz
   ```

   Latest release file is available at Github releases page.

CHAPTER THREE

GETTING STARTED

Solving an optimization problem via **sasoptpy** starts with having a running CAS Server. It is possible to model a problem without a server but solving a problem requires access to SAS Viya Optimization solvers.

3.1 Creating a session

sasoptpy uses the CAS connection provided by the swat package. After installation simply use

```
In [1]: from swat import CAS

In [2]: s = CAS(hostname, port, userid, password)
```

The last two parameters are optional for some cases. See swat Documentation for more details.

3.1.1 Creating a SAS 9.4 session

To create a SAS 9.4 session, see saspy Documentation. After the configurations, a session can be created as follows:

```
import saspy
s = saspy.SASsession(cfgname='winlocal')
```

3.2 Initializing a model

After having an active CAS session, now an empty model can be defined as follows:

```
In [3]: import sasoptpy as so

In [4]: m = so.Model(name='my_first_model', session=s)
NOTE: Initialized model my_first_model.
```

This command creates an empty model.

3.3 Processing input data

The easiest way to work with **sasoptpy** is to define problem inputs as Pandas DataFrames. Objective and cost coefficients, and lower and upper bounds can be defined using the DataFrame and Series objects. See Pandas Documentation to learn more.

```
In [5]: import pandas as pd

In [6]: prob_data = pd.DataFrame([
   ...:     ['Period1', 30, 5],
   ...:     ['Period2', 15, 5],
   ...:     ['Period3', 25, 0]
   ...:     ], columns=['period', 'demand', 'min_prod']).set_index(['period'])
   ...:

In [7]: price_per_product = 10

In [8]: capacity_cost = 10
```

Set `PERIODS` and other fields `demand`, `min_production` can be extracted as follows

```
In [9]: PERIODS = prob_data.index.tolist()

In [10]: demand = prob_data['demand']

In [11]: min_production = prob_data['min_prod']
```

Notice that `PERIODS` is a list, where both `demand` and `min_production` are Pandas Series objects.

3.4 Adding variables

`Model` objects have two different methods for adding variables.

- The first one is *`Model.add_variable()`* which is used to add a single variable.

  ```
  In [12]: production_cap = m.add_variable(vartype=so.INT, name='production_cap',␣
  ↪lb=0)
  ```

  When working with multiple models, you can create a variable independent of the model, such as `production_cap = so.Variable(name='production_cap', vartype=so.INT, lb=0)` and can be added to the model as `m.add_variable(production_cap)`.

- The second one is *`Model.add_variables()`* where a set of variables can be added to the model.

  ```
  In [13]: production = m.add_variables(PERIODS, vartype=so.INT, name='production',
     ....:                              lb=min_production)
     ....:
  ```

  When passed as a set of variables, individual variables can be obtained by using individual keys, such as `production['Period1']`. To create multi-dimensional variables, simply list all the keys as `multivar = m.add_variables(KEYS1, KEYS2, KEYS3, name='multivar')`.

3.5 Creating expressions

`Expression` objects keep linear mathematical expressions. Although these objects are mostly used under the hood when defining a model, it is possible to define a custom *`Expression`* to use later.

```
In [14]: totalRevenue = production.sum('*')*price_per_product

In [15]: totalCost = production_cap * capacity_cost
```

The first thing to notice is the use of the `VariableGroup.sum()` method over a variable group. This method returns the sum of variables inside the group as an `Expression` object. Its multiplication with a scalar `profit_per_product` gives the final expression.

Similarly, `totalCost` is simply multiplication of a `Variable` object with a scalar.

3.6 Setting an objective function

Objective functions can be written in terms of linear expressions. In this problem, the objective is to maximize the profit, so `Model.set_objective()` method is used as follows:

```
In [16]: m.set_objective(totalRevenue-totalCost, sense=so.MAX, name='totalProfit')
Out[16]: sasoptpy.Expression(exp =  10.0 * production['Period2']  -  10.0 *␣
↪production_cap  +  10.0 * production['Period3']  +  10.0 * production['Period1'] ,␣
↪name='obj_1')
```

Notice that you can define the same objective using `m.set_objective(production.sum('*')*price_per_product - production_cap*capacity_cost, sense=so.MAX, name='totalProfit')`

The mandatory argument `sense` should be assigned the value of either `so.MIN` or `so.MAX` for minimization or maximization problems, respectively.

3.7 Adding constraints

In **sasoptpy**, constraints are simply expressions with a direction. It is possible to define an expression and add it to a model by defining which direction the linear relation should have.

There are two methods to add constraints. The first one is `Model.add_constraint()` where a single constraint can be inserted into a model.

The second one is `Model.add_constraints()` where multiple constraints can be added to a model.

```
In [17]: m.add_constraints((production[i] <= production_cap for i in PERIODS),
   ....:                   name='capacity')
   ....:
Out[17]: sasoptpy.ConstraintGroup([ production['Period2']  -  production_cap  <=  0, ␣
↪-  production_cap  +  production['Period1']  <=  0,  -  production_cap  + ␣
↪production['Period3']  <=  0, ], name='capacity')
```

```
In [18]: m.add_constraints((production[i] <= demand[i] for i in PERIODS),
   ....:                   name='demand')
   ....:
Out[18]: sasoptpy.ConstraintGroup([ production['Period2']  <=  15,  production[
↪'Period1']  <=  30,  production['Period3']  <=  25, ], name='demand')
```

Here, the first term provides a Python generator, which then gets translated into constraints in the problem. The symbols <=, >=, and == are used for less than or equal to, greater than or equal to, and equal to constraints, respectively.

3.8 Solving a problem

Defined problems can be simply sent to CAS Servers by calling the `Model.solve()` method.

See the solution output to the problem.

```
In [19]: m.solve()
NOTE: Converting model my_first_model to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPVAOXNI50␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPVAOXNI50 has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem my_first_model has 4 variables (0 binary, 4 integer, 0 free, 0␣
↪fixed).
NOTE: The problem has 6 constraints (6 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 9 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 400.
Out[19]:
Selected Rows from Table PRIMAL

  _OBJ_ID_ _RHS_ID_               _VAR_ _TYPE_  _OBJCOEF_  _LBOUND_  \
0    obj_1      RHS      production_cap      I      -10.0       0.0
1    obj_1      RHS  production_Period1      I       10.0       5.0
2    obj_1      RHS  production_Period2      I       10.0       5.0
3    obj_1      RHS  production_Period3      I       10.0       0.0

        _UBOUND_  _VALUE_  _SOL_
0  1.797693e+308     25.0    1.0
1  1.797693e+308     25.0    1.0
2  1.797693e+308     15.0    1.0
3  1.797693e+308     25.0    1.0
```

At the end of the solve operation, the CAS Server returns both Problem Summary and Solution Summary tables. These tables can be later accessed using `m.get_problem_summary()` and `m.get_solution_summary`.

```
In [20]: print(m.get_solution_summary())
Solution Summary

                                Value
Label
Solver                           MILP
Algorithm               Branch and Cut
Objective Function              obj_1
Solution Status               Optimal
Objective Value                   400

Relative Gap                        0
Absolute Gap                        0
Primal Infeasibility                0
Bound Infeasibility                 0
Integer Infeasibility               0

Best Bound                        400
Nodes                               0
Solutions Found                     1
Iterations                          0
```

(continues on next page)

(continued from previous page)

```
Presolve Time                    0.01
Solution Time                    0.01
```

The *Model.solve()* method returns the primal solution when available, and `None` otherwise.

3.9 Printing solutions

Solutions provided by the solver can be obtained using *sasoptpy.get_solution_table()* method. It is strongly suggested to group variables and expressions that share the same keys in a call.

```
In [21]: print(so.get_solution_table(demand, production))
         demand  production
1
Period1      30        25.0
Period2      15        15.0
Period3      25        25.0
```

As seen, a Pandas Series and a Variable object that has the same index keys are printed in this example.

3.10 Next steps

You can browse *Examples* to see various uses of aformentioned functionality.

If you have a good understanding of the flow, then check *API Reference* to access API details.

CHAPTER

FOUR

HANDLING DATA

sasoptpy can work with native Python types and **pandas** objects for all data operations. Among **pandas** object types, **sasoptpy** works with `pandas.DataFrame` and `pandas.Series` objects to construct and manipulate model components.

4.1 Indices

Methods like *`Model.add_variables()`* can utilize native Python object types like list and range as variable and constraint indices. `pandas.Index` can be used as index as well.

4.1.1 List

```
In [1]: m = so.Model(name='demo')
NOTE: Initialized model demo.

In [2]: SEASONS = ['Fall', 'Winter', 'Spring', 'Summer']

In [3]: prod_lb = {'Fall': 100, 'Winter': 200, 'Spring': 100, 'Summer': 400}

In [4]: production = m.add_variables(SEASONS, lb=prod_lb, name='production')

In [5]: print(production)
Variable Group (production) [
  [Fall: production['Fall']]
  [Spring: production['Spring']]
  [Summer: production['Summer']]
  [Winter: production['Winter']]
]
```

```
In [6]: print(repr(production['Summer']))
sasoptpy.Variable(name='production_Summer', lb=400,  vartype='CONT')
```

Note that if a list is being used as the index set, associated fields like *lb*, *ub* should be accesible using the index keys. Accepted types are dict and pandas.Series.

4.1.2 Range

```
In [7]: link = m.add_variables(range(3), range(2), vartype=so.BIN, name='link')

In [8]: print(link)
Variable Group (link) [
  [(0, 0): link[0, 0]]
  [(0, 1): link[0, 1]]
  [(1, 0): link[1, 0]]
  [(1, 1): link[1, 1]]
  [(2, 0): link[2, 0]]
  [(2, 1): link[2, 1]]
]
```

```
In [9]: print(repr(link[2, 1]))
sasoptpy.Variable(name='link_2_1', ub=1,  vartype='BIN')
```

4.1.3 pandas.Index

```
In [10]: import pandas as pd

In [11]: p_data = [[3, 5, 9],
   ....:            [0, -1, 14],
   ....:            [5, 6, 20]]
   ....:

In [12]: df = pd.DataFrame(p_data, columns=['c1', 'col_lb', 'col_ub'])

In [13]: x = m.add_variables(df.index, lb=df['c1'], vartype=so.INT, name='x')
```

```
In [14]: print(x)
Variable Group (x) [
  [0: x[0]]
  [1: x[1]]
  [2: x[2]]
]
```

```
In [15]: df2 = df.set_index([['r1', 'r2', 'r3']])

In [16]: y = m.add_variables(df2.index, lb=df2['col_lb'], ub=df2['col_ub'], name='y')
```

```
In [17]: print(y)
Variable Group (y) [
  [r1: y['r1']]
  [r2: y['r2']]
  [r3: y['r3']]
]
```

```
In [18]: print(repr(y['r1']))
sasoptpy.Variable(name='y_r1', lb=5, ub=9,  vartype='CONT')
```

4.2 Operations

Lists, `pandas.Series`, and `pandas.DataFrame` objects can be used for mathematical operations like *VariableGroup.mult()*.

```
In [19]: sd = [3, 5, 6]

In [20]: z = m.add_variables(3, name='z')
```

```
In [21]: print(z)
Variable Group (z) [
  [0: z[0]]
  [1: z[1]]
  [2: z[2]]
]
```

```
In [22]: print(repr(z))
sasoptpy.VariableGroup([0, 1, 2], name='z')
```

```
In [23]: e1 = z.mult(sd)

In [24]: print(e1)
 3.0 * z[0]  +  5.0 * z[1]  +  6.0 * z[2]
```

```
In [25]: ps = pd.Series(sd)

In [26]: e2 = z.mult(ps)

In [27]: print(e2)
 3.0 * z[0]  +  5.0 * z[1]  +  6.0 * z[2]
```

CHAPTER

FIVE

SESSIONS AND MODELS

5.1 CAS Sessions

A `swat.cas.connection.CAS` session is needed to solve optimization problems with **sasoptpy**. See SAS documentation to learn more about CAS sessions and SAS Viya.

A sample CAS Session can be created using the following commands.

```
>>> import sasoptpy as so
>>> from swat import CAS
>>> s = CAS(hostname=cas_host, username=cas_username, password=cas_password, port=cas_
↪port)
>>>  m = so.Model(name='demo', session=s)
>>> print(repr(m))
sasoptpy.Model(name='demo', session=CAS(hostname, port, username, protocol='cas',␣
↪name='py-session-1', session=session-no))
```

5.2 Models

5.2.1 Creating a model

An empty model can be created using the *Model* constructor:

```
In [1]: import sasoptpy as so

In [2]: m = so.Model(name='model1')
NOTE: Initialized model model1.
```

5.2.2 Adding new components to a model

Adding a variable:

```
In [3]: x = m.add_variable(name='x', vartype=so.BIN)

In [4]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (1): [
```

(continues on next page)

(continued from previous page)

```
    x
  ]
  Constraints (0): [
  ]
]

In [5]: y = m.add_variable(name='y', lb=1, ub=10)

In [6]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
  Constraints (0): [
  ]
]
```

Adding a constraint:

```
In [7]: c1 = m.add_constraint(x + 2 * y <= 10, name='c1')

In [8]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
     x  +  2.0 * y  <=  10
  ]
]
```

5.2.3 Adding existing components to a model

A new model can use existing variables. The typical way to include a variable is to use the *Model.include()* method:

```
In [9]: new_model = so.Model(name='new_model')
NOTE: Initialized model new_model.

In [10]: new_model.include(x, y)

In [11]: print(new_model)
Model: [
  Name: new_model
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
```

(continues on next page)

(continued from previous page)

```
  Constraints (0): [
  ]
]

In [12]: new_model.include(c1)

In [13]: print(new_model)
Model: [
  Name: new_model
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
    x  +  2.0 * y  <=  10
  ]
]

In [14]: z = so.Variable(name='z', vartype=so.INT, lb=3)

In [15]: new_model.include(z)

In [16]: print(new_model)
Model: [
  Name: new_model
  Objective: MIN []
  Variables (3): [
    x
    y
    z
  ]
  Constraints (1): [
    x  +  2.0 * y  <=  10
  ]
]
```

Note that variables are added to `Model` objects by reference. Therefore, after `Model.solve()` is called, values of variables will be replaced with optimal values.

5.2.4 Accessing components

You can get a list of model variables using `Model.get_variables()` method.

```
In [17]: print(m.get_variables())
[sasoptpy.Variable(name='x', ub=1,  vartype='BIN'), sasoptpy.Variable(name='y', lb=1,
↪ub=10,  vartype='CONT')]
```

Similarly, you can access a list of constraints using `Model.get_constraints()` method.

```
In [18]: c2 = m.add_constraint(2 * x - y >= 1, name='c2')

In [19]: print(m.get_constraints())
[sasoptpy.Constraint( x  +  2.0 * y  <=  10, name='c1'), sasoptpy.Constraint( 2.0 * x
↪ -  y  >=  1, name='c2')]
```

To access a certain constraint using its name, you can use *Model.get_constraint()* method:

```
In [20]: print(m.get_constraint('c2'))
 2.0 * x  -  y  >=  1
```

5.2.5 Dropping components

A variable inside a model can simply be dropped using *Model.drop_variable()*. Similarly, a set of variables can be dropped using *Model.drop_variables()*.

```
In [21]: m.drop_variable(y)

In [22]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (1): [
    x
  ]
  Constraints (2): [
     x  +  2.0 * y  <=  10
     2.0 * x  -  y  >=  1
  ]
]
```

```
In [23]: m.include(y)

In [24]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
  Constraints (2): [
     x  +  2.0 * y  <=  10
     2.0 * x  -  y  >=  1
  ]
]
```

A constraint can be dropped using *Model.drop_constraint()* method. Similarly, a set of constraints can be dropped using *Model.drop_constraints()*.

```
In [25]: m.drop_constraint(c1)

In [26]: m.drop_constraint(c2)

In [27]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
```

(continues on next page)

(continued from previous page)

```
  Constraints (0): [
  ]
]
```

```
In [28]: m.include(c1)

In [29]: print(m)
Model: [
  Name: model1
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
     x  +  2.0 * y  <=  10
  ]
]
```

5.2.6 Copying a model

An exact copy of the existing model can be obtained by including the `Model` object itself.

```
In [30]: copy_model = so.Model(name='copy_model')
NOTE: Initialized model copy_model.

In [31]: copy_model.include(m)

In [32]: print(copy_model)
Model: [
  Name: copy_model
  Objective: MIN []
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
     x  +  2.0 * y  <=  10
  ]
]
```

Note that all variables and constraints are included by reference.

5.2.7 Solving a model

A model is solved using the `Model.solve()` method. This method converts Python definitions into an MPS file and uploads to a CAS server for the optimization action. The type of the optimization problem (Linear Optimization or Mixed Integer Linear Optimization) is determined based on variable types.

```
>>> m.solve()
NOTE: Initialized model model_1
NOTE: Converting model model_1 to DataFrame
NOTE: Added action set 'optimization'.
```

(continues on next page)

(continued from previous page)

```
...
NOTE: Optimal.
NOTE: Objective = 124.343.
NOTE: The Dual Simplex solve time is 0.01 seconds.
```

5.2.8 Solve options

Solver Options

All options listed for the CAS solveLp and solveMilp actions can be used through *Model.solve()* method. LP options can passed to *Model.solve()* using `lp` argument, while MILP options can be passed using `milp` argument:

```
>>> m.solve(milp={'maxtime': 600})
>>> m.solve(lp={'algorithm': 'ipm'})
```

See http://go.documentation.sas.com/?cdcId=vdmmlcdc&cdcVersion=8.11&docsetId=casactmopt&docsetTarget=casactmopt_solvelp_syntax.htm&locale=en for a list of LP options.

See http://go.documentation.sas.com/?cdcId=vdmmlcdc&cdcVersion=8.11&docsetId=casactmopt&docsetTarget=casactmopt_solvemilp_syntax.htm&locale=en for a list of MILP options.

Package Options

Besides `lp` and `milp` arguments, there are 4 arguments that can be passed into *Model.solve()* method:

- name: Upload name of the MPS data
- drop: Option for dropping the data from server after solve
- replace: Option for replacing an existing data with the same name
- primalin: Option for using the current values of the variables as an initial solution

When `primalin` argument is `True`, it grabs *Variable* objects `_init` field. This field can be modified with *Variable.set_init()* method.

5.2.9 Getting solutions

After the solve is completed, all variable and constraint values are parsed automatically. A summary of the problem can be accessed using the *Model.get_problem_summary()* method, and a summary of the solution can be accesed using the *Model.get_solution_summary()* method.

To print values of any object, *get_solution_table()* can be used:

```
>>> print(so.get_solution_table(x, y))
```

All variables and constraints passed into this method are returned based on their indices. See *Examples* for more details.

CHAPTER

SIX

MODEL COMPONENTS

In this part, several model components are discussed with examples. See *Examples* to learn more about how these components can be used to define optimization models.

6.1 Expressions

Expression objects represent linear expressions in **sasoptpy**.

6.1.1 Creating expressions

An *Expression* can be created as follows:

```
In [1]: profit = so.Expression(5 * sales - 3 * material, name='profit')

In [2]: print(repr(profit))
sasoptpy.Expression(exp =  -  3.0 * material  +  5.0 * sales , name='profit')
```

6.1.2 Operations

Getting the current value

After the solve is completed, the current value of an expression can be obtained using the *Expression.get_value()* method:

```
>>> print(profit.get_value())
42.0
```

Getting the dual value

Dual values of *Expression* objects can be obtained using *Variable.get_dual()* and *Constraint.get_dual()* methods.

```
>>> m.solve()
>>> ...
>>> print(x.get_dual())
1.0
```

Addition

There are two ways to add elements to an expression. The first and simpler way creates a new expression at the end:

```
In [3]: tax = 0.5

In [4]: profit_after_tax = profit - tax
```

```
In [5]: print(repr(profit_after_tax))
sasoptpy.Expression(exp =  -  0.5  -  3.0 * material  +  5.0 * sales , name=None)
```

The second way, *Expression.add()* method, takes two arguments: the element to be added and the sign (1 or -1):

```
In [6]: profit_after_tax = profit.add(tax, sign=-1)
```

```
In [7]: print(profit_after_tax)
 -  0.5  -  3.0 * material  +  5.0 * sales
```

```
In [8]: print(repr(profit_after_tax))
sasoptpy.Expression(exp =  -  0.5  -  3.0 * material  +  5.0 * sales , name=None)
```

If the expression is a temporary one, then the addition is performed in place.

Multiplication

You can multiply expressions with scalar values:

```
In [9]: investment = profit.mult(0.2)

In [10]: print(investment)
 -  0.6000000000000001 * material  +  sales
```

Summation

For faster summations compared to Python's native `sum` function, **sasoptpy** provides *sasoptpy.quick_sum()*.

```
In [11]: import time

In [12]: x = m.add_variables(1000, name='x')
```

```
In [13]: t0 = time.time()

In [14]: e = so.quick_sum(2 * x[i] for i in range(1000))

In [15]: print(time.time()-t0)
0.007811307907104492
```

```
In [16]: t0 = time.time()

In [17]: f = sum(2 * x[i] for i in range(1000))

In [18]: print(time.time()-t0)
0.3335609436035156
```

6.1.3 Renaming an expression

Expressions can be renamed using *Expression.set_name()* method:

```
In [19]: e = so.Expression(x[5] + 2 * x[6], name='e1')

In [20]: print(repr(e))
sasoptpy.Expression(exp =  x[5]  +  2.0 * x[6] , name='e1')

In [21]: e.set_name('e2')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[21]: 'e2'

In [22]: print(repr(e))
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\sasoptpy.
↪Expression(exp =  x[5]  +  2.0 * x[6] , name='e2')
```

6.1.4 Copying an expression

An *Expression* can be copied using *Expression.copy()*.

```
In [23]: copy_profit = profit.copy(name='copy_profit')

In [24]: print(repr(copy_profit))
sasoptpy.Expression(exp =  -  3.0 * material  +  5.0 * sales , name='copy_profit')
```

6.1.5 Temporary expressions

An *Expression* object can be defined as temporary, which enables faster `Expression.sum()` and *Expression.mult()* operations.

```
In [25]: new_profit = so.Expression(10 * sales - 2 * material, temp=True)

In [26]: print(repr(new_profit))
sasoptpy.Expression(exp =  -  2.0 * material  +  10.0 * sales , name=None)
```

The expression can be modified inside a function:

```
In [27]: new_profit + 5
Out[27]: sasoptpy.Expression(exp =  5  -  2.0 * material  +  10.0 * sales , name=None)
```

```
In [28]: print(repr(new_profit))
sasoptpy.Expression(exp =  5  -  2.0 * material  +  10.0 * sales , name=None)
```

As you can see, the value of `new_profit` is changed due to an in-place addition. To prevent the change, such expressions can be converted to permanent expressions using the *Expression.set_permanent()* method or constructor:

```
In [29]: new_profit = so.Expression(10 * sales - 2 * material, temp=True)

In [30]: new_profit.set_permanent()
Out[30]: 'expr_1'

In [31]: tmp = new_profit + 5

In [32]: print(repr(new_profit))
sasoptpy.Expression(exp =  -  2.0 * material  +  10.0 * sales , name='expr_1')
```

6.2 Objective Functions

6.2.1 Setting and getting an objective function

Any valid `Expression` can be used as the objective function of a model. An existing expression can be used as an objective function using the `Model.set_objective()` method. The objective function of a model can be obtained using the `Model.get_objective()` method.

```
>>> profit = so.Expression(5 * sales - 2 * material, name='profit')
>>> m.set_objective(profit, so.MAX)
>>> print(m.get_objective())
 -  2.0 * material  +  5.0 * sales
```

6.2.2 Getting the value

After a solve, the objective value can be checked using the `Model.get_objective_value()` method.

```
>>> m.solve()
>>> print(m.get_objective_value())
42.0
```

6.3 Variables

6.3.1 Creating variables

Variables can be created either separately or inside a model.

Creating a variable outside a model

The first way to create a variable uses the default constructor.

```
>>> x = so.Variable(vartype=so.INT, ub=5, name='x')
```

When created separately, a variable needs to be included (or added) inside the model:

```
>>> y = so.Variable(name='y', lb=5)
>>> m.add_variable(y)
```

and

```
>>> y = m.add_variable(name='y', lb=5)
```

are equivalent.

Creating a variable inside a model

The second way is to use `Model.add_variable()`. This method creates a `Variable` object and returns a pointer.

```
>>> x = m.add_variable(vartype=so.INT, ub=5, name='x')
```

6.3.2 Arguments

There are three types of variables: continuous variables, integer variables, and binary variables. Continuous variables are the default type and can be created using the `vartype=so.CONT` argument. Integer variables and binary variables can be created using the `vartype=so.INT` and `vartype=so.BIN` arguments, respectively.

The default lower bound for variables is 0, and the upper bound is infinity. Name is a required argument. If the given name already exists in the namespace, then a different generic name can be used for the variable. The `reset_globals()` function can be used to reset sasoptpy namespace when needed.

6.3.3 Changing bounds

The `Variable.set_bounds()` method changes the bounds of a variable.

```
>>> x = so.Variable(name='x', lb=0, ub=20)
>>> print(repr(x))
sasoptpy.Variable(name='x', lb=0, ub=20, vartype='CONT')
>>> x.set_bounds(lb=5, ub=15)
>>> print(repr(x))
sasoptpy.Variable(name='x', lb=5, ub=15, vartype='CONT')
```

6.3.4 Setting initial values

Initial values of variables can be passed to the solvers for certain problems. The `Variable.set_init()` method changes the initial value for variables. This value can be set at the creation of the variable as well.

```
>>> x.set_init(5)
>>> print(repr(x))
sasoptpy.Variable(name='x', ub=20, init=5,  vartype='CONT')
```

6.3.5 Working with a set of variables

A set of variables can be added using single or multiple indices. Valid index sets include list, dict, and `pandas.Index` objects. See *Handling Data* for more about allowed index types.

Creating a set of variables outside a model

```
>>> production = VariableGroup(PERIODS, vartype=so.INT, name='production',
                               lb=min_production)
>>> print(repr(production))
sasoptpy.VariableGroup(['Period1', 'Period2', 'Period3'], name='production')
>>> m.include(production)
```

Creating a set of variables inside a model

```
>>> production = m.add_variables(PERIODS, vartype=so.INT,
                                 name='production', lb=min_production)
>>> print(production)
>>> print(repr(production))
Variable Group (production) [
  [Period1: production['Period1',]]
  [Period2: production['Period2',]]
  [Period3: production['Period3',]]
```

(continues on next page)

(continued from previous page)

```
]
sasoptpy.VariableGroup(['Period1', 'Period2', 'Period3'],
name='production')
```

6.4 Constraints

6.4.1 Creating constraints

Similar to `Variable` objects, `Constraint` objects can be created inside or outside optimization models.

Creating a constraint outside a model

```
>>> c1 = sasoptpy.Constraint(3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
```

Creating a constraint inside a model

```
>>> c1 = m.add_constraint(3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
```

6.4.2 Modifying variable coefficients

The coefficient of a variable inside a constraint can be updated using the `Constraint.update_var_coef()` method:

```
>>> c1 = so.Constraint(exp=3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
>>> c1.update_var_coef(x, -1)
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  -  x  <=  10, name='c1')
```

6.4.3 Working with a set of constraints

A set of constraints can be added using single or multiple indices. Valid index sets include list, dict, and `pandas.Index` objects. See *Handling Data* for more about allowed index types.

Creating a set of variables outside a model

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z', lb=0, ub=10)
>>> cg = so.ConstraintGroup((2 * z[i, j] + 3 * z[i-1, j] >= 2 for i in
                             [1] for j in ['a', 'b', 'c']), name='cg')
>>> print(cg)
Constraint Group (cg) [
  [(1, 'a'):  3.0 * z[0, 'a']  +  2.0 * z[1, 'a']  >=  2]
  [(1, 'b'):  3.0 * z[0, 'b']  +  2.0 * z[1, 'b']  >=  2]
  [(1, 'c'):  2.0 * z[1, 'c']  +  3.0 * z[0, 'c']  >=  2]
]
```

Creating a set of variables inside a model

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z', lb=0, ub=10)
>>> cg2 = m.add_constraints((2 * z[i, j] + 3 * z[i-1, j] >= 2 for i in
                              [1] for j in ['a', 'b', 'c']), name='cg2')
>>> print(cg2)
Constraint Group (cg2) [
  [(1, 'a'):  2.0 * z[1, 'a']  +  3.0 * z[0, 'a']  >=  2]
  [(1, 'b'):  3.0 * z[0, 'b']  +  2.0 * z[1, 'b']  >=  2]
  [(1, 'c'):  2.0 * z[1, 'c']  +  3.0 * z[0, 'c']  >=  2]
]
```

CHAPTER

SEVEN

API REFERENCE

7.1 Classes

Model(name[, session])	Creates an optimization model
Expression([exp, name, temp])	Creates a linear expression to represent model components
Variable(name[, vartype, lb, ub, init])	Creates an optimization variable to be used inside models
VariableGroup(*argv, name[, vartype, lb, . . .])	Creates a group of *Variable* objects
Constraint(exp[, direction, name, crange])	Creates a linear or quadratic constraint for optimization models
ConstraintGroup(argv, name)	Creates a group of *Constraint* objects

7.1.1 sasoptpy.Model

class sasoptpy.**Model**(*name*, *session=None*)

Creates an optimization model

Parameters **name** : string

Name of the model

session : `swat.cas.connection.CAS` object or `saspy.SASsession` object, optional

CAS or SAS Session object

Examples

```
>>> from swat import CAS
>>> import sasoptpy as so
>>> s = CAS('cas.server.address', port=12345)
>>> m = so.Model(name='my_model', session=s)
NOTE: Initialized model my_model
```

```
>>> mip = so.Model(name='mip')
NOTE: Initialized model mip
```

Methods

add_constraint(c[, name])	Adds a single constraint to the model
add_constraints(argv[, cg, name])	Adds a set of constraints to the model
add_variable([var, vartype, name, lb, ub, init])	Adds a new variable to the model
add_variables(*argv[, vg, name, vartype, . . .])	Adds a group of variables to the model
drop_constraint(constraint)	Drops a constraint from the model
drop_constraints(constraints)	Drops a constraint group from the model
drop_variable(variable)	Drops a variable from the model
drop_variables(variables)	Drops a variable group from the model
get_constraint(name)	Returns the reference to a constraint in the model
get_constraints()	Returns a list of constraints in the model
get_objective()	Returns the objective function as an Expression object
get_objective_value()	Returns the optimal objective value, if it exists
get_problem_summary()	Returns the problem summary table to the user
get_solution([vtype])	Returns the solution details associated with the primal or dual solution
get_solution_summary()	Returns the solution summary table to the user
get_variable(name)	Returns the reference to a variable in the model
get_variable_coef(var)	Returns the objective value coefficient of a variable
get_variables()	Returns a list of variables
include(*argv)	Adds existing variables and constraints to a model
print_solution()	Prints the current values of the variables
set_coef(var, con, value)	Updates the coefficient of a variable inside constraints
set_objective(expression, sense[, name])	Sets the objective function for the model
set_session(session)	Sets the CAS session for model
solve([milp, lp, name, drop, replace, primalin])	Solves the model by calling CAS optimization solvers
solve_local([name])	(Experimental) Solves the model by calling SAS 9.4 solvers
test_session()	Tests if the model session is defined and still active
to_frame()	Converts the Python model into a DataFrame object in MPS format
upload_model([name, replace])	Converts internal model to MPS table and upload to CAS session
upload_user_blocks()	Uploads user-defined decomposition blocks to the CAS server

sasoptpy.Model.add_constraint

Model.**add_constraint**(*c*, *name=None*)

Adds a single constraint to the model

Parameters **c** : Constraint

Constraint to be added to the model

name : string, optional

Name of the constraint

Returns Constraint object

Examples

```
>>> x = m.add_variable(name='x', vartype=so.INT, lb=0, ub=5)
>>> y = m.add_variables(3, name='y', vartype=so.CONT, lb=0, ub=10)
>>> c1 = m.add_constraint(x + y[0] >= 3, name='c1')
>>> print(c1)
 x  +  y[0]  >=  3
```

```
>>> c2 = m.add_constraint(x - y[2] == [4, 10], name='c2')
>>> print(c2)
 -  y[2]  +  x  =  [4, 10]
```

sasoptpy.Model.add_constraints

`Model.add_constraints`(*argv*, *cg=None*, *name=None*)

Adds a set of constraints to the model

Parameters **argv** : Generator type objects

List of constraints as a Generator-type object

cg : *ConstraintGroup* object, optional

An existing list of constraints if an existing group is being added

name : string, optional

Name for the constraint group and individual constraint prefix

Returns *ConstraintGroup* object

A group object for all constraints aded

Examples

```
>>> x = m.add_variable(name='x', vartype=so.INT, lb=0, ub=5)
>>> y = m.add_variables(3, name='y', vartype=so.CONT, lb=0, ub=10)
>>> c = m.add_constraints((x + 2 * y[i] >= 2 for i in [0, 1, 2]),
                          name='c')
>>> print(c)
Constraint Group (c) [
  [0:  2.0 * y[0]  +  x  >=  2]
  [1:  2.0 * y[1]  +  x  >=  2]
  [2:  2.0 * y[2]  +  x  >=  2]
]
```

```
>>> t = m.add_variables(3, 4, name='t')
>>> ct = m.add_constraints((t[i, j] <= x for i in range(3)
                            for j in range(4)), name='ct')
>>> print(ct)
Constraint Group (ct) [
  [(0, 0):  -  x  +  t[0, 0]  <=  0]
  [(0, 1):  t[0, 1]  -  x  <=  0]
  [(0, 2):  -  x  +  t[0, 2]  <=  0]
  [(0, 3):  t[0, 3]  -  x  <=  0]
  [(1, 0):  t[1, 0]  -  x  <=  0]
```

(continues on next page)

(continued from previous page)

```
  [(1, 1):  t[1, 1]  -  x  <=  0]
  [(1, 2):  -  x  +  t[1, 2]  <=  0]
  [(1, 3):  -  x  +  t[1, 3]  <=  0]
  [(2, 0):  -  x  +  t[2, 0]  <=  0]
  [(2, 1):  t[2, 1]  -  x  <=  0]
  [(2, 2):  t[2, 2]  -  x  <=  0]
  [(2, 3):  t[2, 3]  -  x  <=  0]
]
```

sasoptpy.Model.add_variable

`Model.`**`add_variable`**(*var=None*, *vartype='CONT'*, *name=None*, *lb=0*, *ub=inf*, *init=None*)

Adds a new variable to the model

New variables can be created via this method or existing variables can be added to the model.

Parameters **var** : `Variable` object, optional

Existing variable to be added to the problem

vartype : string, optional

Type of the variable, either 'BIN', 'INT' or 'CONT'

name : string, optional

Name of the variable to be created

lb : float, optional

Lower bound of the variable

ub : float, optional

Upper bound of the variable

init : float, optional

Initial value of the variable

Returns `Variable` object

Variable that is added to the model

See also:

`Model.include()`

Notes

- If argument *var* is not None, then all other arguments are ignored.
- A generic variable name is generated if name argument is None.

Examples

Adding a variable on the fly

```
>>> m = so.Model(name='demo')
>>> x = m.add_variable(name='x', vartype=so.INT, ub=10, init=2)
>>> print(repr(x))
NOTE: Initialized model demo
sasoptpy.Variable(name='x', lb=0, ub=10, init=2, vartype='INT')
```

Adding an existing variable to a model

```
>>> y = so.Variable(name='y', vartype=so.BIN)
>>> m = so.Model(name='demo')
>>> m.add_variable(var=y)
```

sasoptpy.Model.add_variables

`Model.`**`add_variables`**(**argv*, *vg=None*, *name=None*, *vartype='CONT'*, *lb=None*, *ub=None*, *init=None*)

Adds a group of variables to the model

Parameters **argv** : list, dict, `pandas.Index`

Loop index for variable group

vg : *`VariableGroup`* object, optional

An existing object if it is being added to the model

name : string, optional

Name of the variables

vartype : string, optional

Type of variables, *BIN*, *INT*, or *CONT*

lb : list, dict, `pandas.Series`

Lower bounds of variables

ub : list, dict, `pandas.Series`

Upper bounds of variables

init : list, dict, `pandas.Series`

Initial values of variables

See also:

`VariableGroup`

Notes

If *vg* argument is passed, all other arguments are ignored.

Examples

```
>>> production = m.add_variables(PERIODS, vartype=so.INT,
                                 name='production', lb=min_production)
>>> print(production)
>>> print(repr(production))
Variable Group (production) [
  [Period1: production['Period1',]]
  [Period2: production['Period2',]]
  [Period3: production['Period3',]]
]
sasoptpy.VariableGroup(['Period1', 'Period2', 'Period3'],
name='production')
```

sasoptpy.Model.drop_constraint

`Model.drop_constraint(constraint)`

Drops a constraint from the model

Parameters constraint : `Constraint` object

The constraint to be dropped from the model

See also:

`Model.drop_constraints()`, `Model.drop_variable()`, `Model.drop_variables()`

Examples

```
>>> c1 = m.add_constraint(2 * x + y <= 15, name='c1')
>>> print(m.get_constraint('c1'))
  2.0 * x  +  y  <=  15
>>> m.drop_constraint(c1)
>>> print(m.get_constraint('c1'))
None
```

sasoptpy.Model.drop_constraints

`Model.drop_constraints(constraints)`

Drops a constraint group from the model

Parameters constraints : `ConstraintGroup` object

The constraint group to be dropped from the model

See also:

`Model.drop_constraints()`, `Model.drop_variable()`, `Model.drop_variables()`

Examples

```
>>> c1 = m.add_constraints((x[i] + y <= 15 for i in [0, 1]), name='c1')
>>> print(m.get_constraints())
[sasoptpy.Constraint( x[0]  +  y  <=  15, name='c1_0'),
 sasoptpy.Constraint( x[1]  +  y  <=  15, name='c1_1')]
```

(continues on next page)

(continued from previous page)

```
>>> m.drop_constraints(c1)
>>> print(m.get_constraints())
[]
```

sasoptpy.Model.drop_variable

Model.**drop_variable**(*variable*)

Drops a variable from the model

Parameters variable : *Variable* object

The variable to be dropped from the model

See also:

Model.drop_variables(), *Model.drop_constraint()*, *Model.drop_constraints()*

Examples

```
>>> x = m.add_variable(name='x')
>>> y = m.add_variable(name='y')
>>> print(m.get_variable('x'))
 x
>>> m.drop_variable(x)
>>> print(m.get_variable('x'))
None
```

sasoptpy.Model.drop_variables

Model.**drop_variables**(*variables*)

Drops a variable group from the model

Parameters variables : *VariableGroup* object

The variable group to be dropped from the model

See also:

Model.drop_variable(), *Model.drop_constraint()*, *Model.drop_constraints()*

Examples

```
>>> x = m.add_variables(3, name='x')
>>> print(m.get_variables())
[sasoptpy.Variable(name='x_0',  vartype='CONT'),
 sasoptpy.Variable(name='x_1',  vartype='CONT')]
>>> m.drop_variables(x)
>>> print(m.get_variables())
[]
```

sasoptpy.Model.get_constraint

`Model.get_constraint(name)`

Returns the reference to a constraint in the model

Parameters name : string

Name of the constraint requested

Returns `Constraint` object

Examples

```
>>> m.add_constraint(2 * x + y <= 15, name='c1')
>>> print(m.get_constraint('c1'))
2.0 * x  +  y  <=  15
```

sasoptpy.Model.get_constraints

`Model.get_constraints()`

Returns a list of constraints in the model

Returns list : A list of `Constraint` objects

Examples

```
>>> m.add_constraint(x[0] + y <= 15, name='c1')
>>> m.add_constraints((2 * x[i] - y >= 1 for i in [0, 1]), name='c2')
>>> print(m.get_constraints())
[sasoptpy.Constraint( x[0]  +  y  <=  15, name='c1'),
 sasoptpy.Constraint( 2.0 * x[0]  -  y  >=  1, name='c2_0'),
 sasoptpy.Constraint( 2.0 * x[1]  -  y  >=  1, name='c2_1')]
```

sasoptpy.Model.get_objective

`Model.get_objective()`

Returns the objective function as an `Expression` object

Returns `Expression` object

Objective function

Examples

```
>>> m.set_objective(4 * x - 5 * y, name='obj')
>>> print(repr(m.get_objective()))
sasoptpy.Expression(exp =  4.0 * x  -  5.0 * y , name='obj')
```

sasoptpy.Model.get_objective_value

Model.**get_objective_value**()

Returns the optimal objective value, if it exists

Returns **float** : Objective value at current solution

Examples

```
>>> m.solve()
>>> print(m.get_objective_value())
42.0
```

sasoptpy.Model.get_problem_summary

Model.**get_problem_summary**()

Returns the problem summary table to the user

Returns `swat.dataframe.SASDataFrame` object

Problem summary obtained after *Model.solve()*

Examples

```
>>> m.solve()
>>> ps = m.get_problem_summary()
>>> print(type(ps))
<class 'swat.dataframe.SASDataFrame'>
```

```
>>> print(ps)
Problem Summary
                                Value
Label
Problem Name                   model1
Objective Sense          Maximization
Objective Function                obj
RHS                               RHS
Number of Variables                 2
Bounded Above                       0
Bounded Below                       2
Bounded Above and Below             0
Free                                0
Fixed                               0
Number of Constraints               2
LE (<=)                             1
EQ (=)                              0
GE (>=)                             1
Range                               0
Constraint Coefficients             4
```

```
>>> print(ps.index)
Index(['Problem Name', 'Objective Sense', 'Objective Function', 'RHS',
'', 'Number of Variables', 'Bounded Above', 'Bounded Below',
```

(continues on next page)

(continued from previous page)

```
'Bounded Above and Below', 'Free', 'Fixed', '',
'Number of Constraints', 'LE (<=)', 'EQ (=)', 'GE (>=)', 'Range', '',
'Constraint Coefficients'],
dtype='object', name='Label')
```

```
>>> print(ps.loc['Number of Variables'])
Value           2
Name: Number of Variables, dtype: object
```

```
>>> print(ps.loc['Constraint Coefficients', 'Value'])
4
```

sasoptpy.Model.get_solution

`Model.get_solution(vtype='Primal')`

Returns the solution details associated with the primal or dual solution

Parameters **vtype** : string, optional

'Primal' or 'Dual'

Returns `pandas.DataFrame` object

Primal or dual solution table returned from the CAS Action

Examples

```
>>> m.solve()
>>> print(m.get_solution('Primal'))
     _OBJ_ID_ _RHS_ID_               _VAR_ _TYPE_  _OBJCOEF_  _LBOUND_
0  totalProfit      RHS      production_cap      I      -10.0       0.0
1  totalProfit      RHS  production_Period1      I       10.0       5.0
2  totalProfit      RHS  production_Period2      I       10.0       5.0
3  totalProfit      RHS  production_Period3      I       10.0       0.0
      _UBOUND_  _VALUE_
1.797693e+308     25.0
1.797693e+308     25.0
1.797693e+308     15.0
1.797693e+308     25.0
```

```
>>> print(m.get_solution('Dual'))
     _OBJ_ID_ _RHS_ID_       _ROW_ _TYPE_  _RHS_  _L_RHS_  _U_RHS_
0  totalProfit      RHS  capacity_0      L    0.0      NaN      NaN
1  totalProfit      RHS  capacity_1      L    0.0      NaN      NaN
2  totalProfit      RHS  capacity_2      L    0.0      NaN      NaN
3  totalProfit      RHS    demand_0      L   30.0      NaN      NaN
4  totalProfit      RHS    demand_1      L   15.0      NaN      NaN
5  totalProfit      RHS    demand_2      L   25.0      NaN      NaN
_ACTIVITY_
       0.0
     -10.0
       0.0
      25.0
```

(continues on next page)

(continued from previous page)

```
      15.0
      25.0
```

sasoptpy.Model.get_solution_summary

`Model.get_solution_summary()`

Returns the solution summary table to the user

Returns `swat.dataframe.SASDataFrame` object

Solution summary obtained after solve

Examples

```
>>> m.solve()
>>> soln = m.get_solution_summary()
>>> print(type(soln))
<class 'swat.dataframe.SASDataFrame'>
```

```
>>> print(soln)
Solution Summary
                                  Value
Label
Solver                               LP
Algorithm                  Dual Simplex
Objective Function                  obj
Solution Status                 Optimal
Objective Value                      10
Primal Infeasibility                  0
Dual Infeasibility                    0
Bound Infeasibility                   0
Iterations                            2
Presolve Time                      0.00
Solution Time                      0.01
```

```
>>> print(soln.index)
Index(['Solver', 'Algorithm', 'Objective Function', 'Solution Status',
       'Objective Value', '', 'Primal Infeasibility',
       'Dual Infeasibility', 'Bound Infeasibility', '', 'Iterations',
       'Presolve Time', 'Solution Time'],
      dtype='object', name='Label')
```

```
>>> print(soln.loc['Solution Status', 'Value'])
Optimal
```

sasoptpy.Model.get_variable

`Model.get_variable(name)`

Returns the reference to a variable in the model

Parameters **name** : string

Name or key of the variable requested

Returns *Variable* object

Examples

```
>>> m.add_variable(name='x', vartype=so.INT, lb=3, ub=5)
>>> var1 = m.get_variable('x')
>>> print(repr(var1))
sasoptpy.Variable(name='x', lb=3, ub=5, vartype='INT')
```

sasoptpy.Model.get_variable_coef

`Model.get_variable_coef(var)`

Returns the objective value coefficient of a variable

Parameters **var** : *Variable* object or string

Variable whose objective value is requested or its name

Returns float

Objective value coefficient of the given variable

Examples

```
>>> x = m.add_variable(name='x')
>>> y = m.add_variable(name='y')
>>> m.set_objective(4 * x - 5 * y, name='obj', sense=so.MAX)
>>> print(m.get_variable_coef(x))
4.0
>>> print(m.get_variable_coef('y'))
-5.0
```

sasoptpy.Model.get_variables

`Model.get_variables()`

Returns a list of variables

Returns **list** : A list of *Variable* objects

Examples

```
>>> x = m.add_variables(2, name='x')
>>> y = m.add_variable(name='y')
>>> print(m.get_variables())
[sasoptpy.Variable(name='x_0',  vartype='CONT'),
 sasoptpy.Variable(name='x_1',  vartype='CONT'),
 sasoptpy.Variable(name='y',  vartype='CONT')]
```

sasoptpy.Model.include

`Model.include(*argv)`

Adds existing variables and constraints to a model

Parameters argv : `Model`, `Variable`, `Constraint`,

`VariableGroup`, `ConstraintGroup` Objects to be included in the model

Notes

- This method is essentially a wrapper for two methods, `Model.add_variable()` and `Model.add_constraint()`.
- Including a model causes all variables and constraints inside the original model to be included.

Examples

Adding an existing variable

```
>>> x = so.Variable(name='x', vartype=so.CONT)
>>> m.include(x)
```

Adding an existing constraint

```
>>> c1 = so.Constraint(x + y <= 5, name='c1')
>>> m.include(c1)
```

Adding an existing set of variables

```
>>> z = so.VariableGroup(3, 5, name='z', ub=10)
>>> m.include(z)
```

Adding an existing set of constraints

```
>>> c2 = so.ConstraintGroup((x + 2 * z[i, j] >= 2 for i in range(3)
                            for j in range(5)), name='c2')
>>> m.include(c2)
```

Adding an existing model (including its elements)

```
>>> new_model = so.Model(name='new_model')
>>> new_model.include(m)
```

sasoptpy.Model.print_solution

`Model.print_solution()`

Prints the current values of the variables

See also:

`Model.get_solution()`

Examples

```
>>> m.solve()
>>> m.print_solution()
x: 2.0
y: 0.0
```

sasoptpy.Model.set_coef

Model.**set_coef**(*var*, *con*, *value*)

Updates the coefficient of a variable inside constraints

Parameters **var** : `Variable` object

Variable whose coefficient will be updated

con : `Constraint` object

Constraint where the coefficient will be updated

value : float

The new value for the coefficient of the variable

See also:

`Constraint.update_var_coef()`

Notes

Variable coefficient inside the constraint is replaced in-place.

Examples

```
>>> c1 = m.add_constraint(x + y >= 1, name='c1')
>>> print(c1)
y  +  x  >=  1
>>> m.set_coef(x, c1, 3)
>>> print(c1)
y  +  3.0 * x  >=  1
```

sasoptpy.Model.set_objective

Model.**set_objective**(*expression*, *sense*, *name=None*)

Sets the objective function for the model

Parameters **expression** : `Expression` object

The objective function as an Expression

sense : string

Objective value direction, 'MIN' or 'MAX'

name : string, optional

Name of the objective value

Returns *Expression*

Objective function as an *Expression* object

Examples

```
>>> profit = so.Expression(5 * sales - 2 * material, name='profit')
>>> m.set_objective(profit, so.MAX)
>>> print(m.get_objective())
 -  2.0 * material  +  5.0 * sales
```

```
>>> m.set_objective(4 * x - 5 * y, name='obj')
>>> print(repr(m.get_objective()))
sasoptpy.Expression(exp =  4.0 * x  -  5.0 * y , name='obj')
```

sasoptpy.Model.set_session

Model.**set_session**(*session*)

Sets the CAS session for model

Parameters **session** : swat.cas.connection.CAS or saspy.SASsession objects

CAS or SAS Session object

Notes

- Session of a model can be set at initialization. See *Model*.

sasoptpy.Model.solve

Model.**solve**(*milp={}*, *lp={}*, *name=None*, *drop=False*, *replace=True*, *primalin=False*)

Solves the model by calling CAS optimization solvers

Parameters **milp** : dict, optional

A dictionary of MILP options for the solveMilp CAS Action

lp : dict, optional

A dictionary of LP options for the solveLp CAS Action

name : string, optional

Name of the table name on CAS Server

drop : boolean, optional

Switch for dropping the MPS table on CAS Server after solve

replace : boolean, optional

Switch for replacing an existing MPS table on CAS Server

primalin : boolean, optional

Switch for using initial values (for MIP only)

Returns `pandas.DataFrame` object

Solution of the optimization model

See also:

Model.solve_local()

Notes

- This method takes two optional arguments (milp and lp).
- These arguments pass options to the solveLp and solveMilp CAS actions.
- These arguments are not passed if the model has a SAS session.
- Both milp and lp should be defined as dictionaries, where keys are option names. For example, `m.solve(milp={'maxtime': 600})` limits solution time to 600 seconds.
- See http://go.documentation.sas.com/?cdcId=vdmmlcdc&cdcVersion=8.11&docsetId=casactmopt&docsetTarget=casactmopt_solvelp_syntax.htm&locale=en for a list of LP options.
- See http://go.documentation.sas.com/?cdcId=vdmmlcdc&cdcVersion=8.11&docsetId=casactmopt&docsetTarget=casactmopt_solvemilp_syntax.htm&locale=en for a list of MILP options.

Examples

```
>>> m.solve()
NOTE: Initialized model food_manufacture_1
NOTE: Converting model food_manufacture_1 to DataFrame
NOTE: Added action set 'optimization'.
...
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Dual Simplex solve time is 0.01 seconds.
```

```
>>> m.solve(milp={'maxtime': 600})
```

```
>>> m.solve(lp={'algorithm': 'ipm'})
```

sasoptpy.Model.solve_local

`Model.`**`solve_local`**`(`*name='MPS'*`)`

(Experimental) Solves the model by calling SAS 9.4 solvers

Parameters **name** : string, optional

Name of the MPS table

See also:

Model.solve()

Notes

- If the session of a model is a `saspy.SASsession` object, then *`Model.solve()`* calls this method internally.
- To use this method, you need to have saspy installed on your Python environment.
- This function is experimental.
- Unlike *`Model.solve()`*, this method does not accept LP and MILP options yet.

Examples

```
>>> import saspy
>>> import sasoptpy as so
>>> sas = saspy.SASsession(cfgname='winlocal')
>>> m = so.Model(name='demo', session=sas)
>>> choco = m.add_variable(lb=0, ub=20, name='choco', vartype=so.INT)
>>> toffee = m.add_variable(lb=0, ub=30, name='toffee')
>>> m.set_objective(0.25*choco + 0.75*toffee, sense=so.MAX, name='profit')
>>> m.add_constraint(15*choco + 40*toffee <= 27000, name='process1')
>>> m.add_constraint(56.25*toffee <= 27000, name='process2')
>>> m.add_constraint(18.75*choco <= 27000, name='process3')
>>> m.add_constraint(12*choco + 50*toffee <= 27000, name='process4')
>>>
>>> m.solve_local()
>>> # or m.solve()
SAS Connection established. Subprocess id is 18192
NOTE: Initialized model demo.
NOTE: Converting model demo to DataFrame.
NOTE: Writing HTML5(SASPY_INTERNAL) Body file: _TOMODS1
NOTE: The problem demo has 2 variables (0 binary, 1 integer, 0 free, 0 fixed).
NOTE: The problem has 4 constraints (4 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 6 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: Optimal.
NOTE: Objective = 27.5.
NOTE: The data set WORK.PROB_SUMMARY has 21 observations and 3 variables.
NOTE: The data set WORK.SOL_SUMMARY has 17 observations and 3 variables.
NOTE: There were 23 observations read from the data set WORK.MPS.
NOTE: The data set WORK.PRIMAL_OUT has 2 observations and 8 variables.
NOTE: The data set WORK.DUAL_OUT has 4 observations and 8 variables.
NOTE: PROCEDURE OPTMILP used (Total process time):
real time           0.07 seconds
cpu time            0.04 seconds
SAS Connection terminated. Subprocess id was 18192
```

sasoptpy.Model.test_session

`Model.`**`test_session`**`()`

Tests if the model session is defined and still active

Returns string

'CAS' for CAS sessions, 'SAS' for SAS sessions, None otherwise

sasoptpy.Model.to_frame

`Model.to_frame()`

Converts the Python model into a DataFrame object in MPS format

Returns `pandas.DataFrame` object

Problem in strict MPS format

Notes

- This method is called inside *Model.solve()*.

Examples

```
>>> df = m.to_frame()
>>> print(df)
     Field1 Field2  Field3 Field4 Field5 Field6 _id_
0      NAME         model1      0             0    1
1      ROWS                                        2
2       MAX    obj                                 3
3         L     c1                                 4
4   COLUMNS                                        5
5                x     obj      4                  6
6                x      c1      3                  7
7                y     obj     -5                  8
8                y      c1      1                  9
9       RHS                                       10
10             RHS      c1      6                 11
11   RANGES                                       12
12   BOUNDS                                       13
13   ENDATA                     0             0   14
```

sasoptpy.Model.upload_model

`Model.upload_model(`*name=None*, *replace=True*`)`

Converts internal model to MPS table and upload to CAS session

Parameters **name** : string, optional

Desired name of the MPS table on the server

replace : boolean, optional

Option to replace the existing MPS table

Returns `swat.cas.table.CASTable` object

Reference to the uploaded CAS Table

Notes

- This method returns None if the model session is not valid.
- Name of the table is randomly assigned if name argument is None or not given.

- This method should not be used if *Model.solve()* is going to be used. *Model.solve()* calls this method internally.

sasoptpy.Model.upload_user_blocks

Model.**upload_user_blocks**()

Uploads user-defined decomposition blocks to the CAS server

Returns string

CAS table name of the user-defined decomposition blocks

Examples

```
>>> userblocks = m.upload_user_blocks()
>>> m.solve(milp={'decomp': {'blocks': userblocks}})
```

7.1.2 sasoptpy.Expression

class sasoptpy.**Expression**(*exp=None*, *name=None*, *temp=False*)

Creates a linear expression to represent model components

Parameters **exp** : *Expression* object, optional

An existing expression where arguments are being passed

name : string, optional

A local name for the expression

temp : boolean, optional

A boolean shows whether expression is temporary or permanent

Notes

- Two other classes (*Variable* and *Constraint*) are subclasses of this class.
- Expressions are created automatically after linear math operations with variables.
- An expression object can be called when defining constraints and other expressions.

Examples

```
>>> x = so.Variable(name='x')
>>> y = so.VariableGroup(3, name='y')
>>> e = so.Expression(exp=x + 3 * y[0] - 5 * y[1], name='exp1')
>>> print(e)
 -  5.0 * y[1]  +  3.0 * y[0]  +  x
>>> print(repr(e))
sasoptpy.Expression(exp =  -  5.0 * y[1]  +  3.0 * y[0]  +  x ,
                    name='exp1')
```

```
>>> sales = so.Variable(name='sales')
>>> material = so.Variable(name='material')
>>> profit = 5 * sales - 3 * material
>>> print(profit)
 5.0 * sales  -  3.0 * material
>>> print(repr(profit))
sasoptpy.Expression(exp =  5.0 * sales  -  3.0 * material , name=None)
```

Methods

`add`(other[, sign])	Combines two expressions and produces a new one
`copy`([name])	Returns a copy of the `Expression` object
`get_dual`()	Returns the dual value
`get_name`()	Returns the name of the expression
`get_value`()	Returns the value of the expression after variable values are changed
`mult`(other)	Multiplies the `Expression` with a scalar value
`set_name`(name)	Sets the name of the expression
`set_permanent`([name])	Converts a temporary expression into a permanent one

sasoptpy.Expression.add

`Expression.`**`add`**(*other*, *sign=1*)

Combines two expressions and produces a new one

Parameters **other** : float or `Expression` object

Second expression or constant value to be added

sign : int, optional

Sign of the addition, 1 or -1

in_place : boolean, optional

Whether the addition will be performed in place or not

Returns `Expression` object

Notes

- This method is mainly for internal use.
- Adding an expression is equivalent to calling this method: (x-y)+(3*x-2*y) and (x-y).add(3*x-2*y) are interchangeable.

sasoptpy.Expression.copy

`Expression.`**`copy`**(*name=None*)

Returns a copy of the `Expression` object

Parameters **name** : string, optional

Name for the copy

Returns *Expression* object

Copy of the object

Examples

```
>>> e = so.Expression(7 * x - y[0], name='e')
>>> print(repr(e))
sasoptpy.Expression(exp =  -  y[0]  +  7.0 * x , name='e')
>>> f = e.copy(name='f')
>>> print(repr(f))
sasoptpy.Expression(exp =  -  y[0]  +  7.0 * x , name='f')
```

sasoptpy.Expression.get_dual

`Expression.get_dual()`

Returns the dual value

Returns float

Dual value of the variable

sasoptpy.Expression.get_name

`Expression.get_name()`

Returns the name of the expression

Returns string

Name of the expression

Examples

```
>>> var1 = m.add_variables(name='x')
>>> print(var1.get_name())
x
```

sasoptpy.Expression.get_value

`Expression.get_value()`

Returns the value of the expression after variable values are changed

Returns float

Value of the expression

Examples

```
>>> profit = so.Expression(5 * sales - 3 * material)
>>> m.solve()
>>> print(profit.get_value())
41.0
```

sasoptpy.Expression.mult

Expression.**mult**(*other*)
Multiplies the *Expression* with a scalar value

Parameters **other** : *Expression* or int

Second expression to be multiplied

Returns *Expression* object

A new *Expression* that represents the multiplication

Notes

- This method is mainly for internal use.
- Multiplying an expression is equivalent to calling this method: 3*(x-y) and (x-y).mult(3) are interchangeable.

sasoptpy.Expression.set_name

Expression.**set_name**(*name*)
Sets the name of the expression

Parameters **name** : string

Name of the expression

Returns string

Name of the expression after resolving conflicts

Examples

```
>>> e = x + 2*y
>>> e.set_name('objective')
```

sasoptpy.Expression.set_permanent

Expression.**set_permanent**(*name=None*)
Converts a temporary expression into a permanent one

Parameters **name** : string, optional

Name of the expression

Returns string

Name of the expression in the namespace

7.1.3 sasoptpy.Variable

class sasoptpy.**Variable**(*name*, *vartype='CONT'*, *lb=0*, *ub=inf*, *init=None*)

Creates an optimization variable to be used inside models

Parameters **name** : string

Name of the variable

vartype : string, optional

Type of the variable

lb : float, optional

Lower bound of the variable

ub : float, optional

Upper bound of the variable

init : float, optional

Initial value of the variable

See also:

sasoptpy.Model.add_variable()

Examples

```
>>> x = so.Variable(name='x', lb=0, ub=20, vartype=so.CONT)
>>> print(repr(x))
sasoptpy.Variable(name='x', lb=0, ub=20, vartype='CONT')
```

```
>>> y = so.Variable(name='y', init=1, vartype=so.INT)
>>> print(repr(y))
sasoptpy.Variable(name='y', lb=0, ub=inf, init=1, vartype='INT')
```

Methods

add(other[, sign])	Combines two expressions and produces a new one
copy([name])	Returns a copy of the *Expression* object
get_dual()	Returns the dual value
get_name()	Returns the name of the expression
get_value()	Returns the value of the expression after variable values are changed
mult(other)	Multiplies the *Expression* with a scalar value
set_bounds([lb, ub])	Changes bounds on a variable
set_init([init])	Changes initial value of a variable
set_name(name)	Sets the name of the expression
set_permanent([name])	Converts a temporary expression into a permanent one

sasoptpy.Variable.add

`Variable.add(other, sign=1)`

Combines two expressions and produces a new one

Parameters **other** : float or *Expression* object

Second expression or constant value to be added

sign : int, optional

Sign of the addition, 1 or -1

in_place : boolean, optional

Whether the addition will be performed in place or not

Returns *Expression* object

Notes

- This method is mainly for internal use.
- Adding an expression is equivalent to calling this method: (x-y)+(3*x-2*y) and (x-y).add(3*x-2*y) are interchangeable.

sasoptpy.Variable.copy

`Variable.copy(name=None)`

Returns a copy of the *Expression* object

Parameters **name** : string, optional

Name for the copy

Returns *Expression* object

Copy of the object

Examples

```
>>> e = so.Expression(7 * x - y[0], name='e')
>>> print(repr(e))
sasoptpy.Expression(exp =  -  y[0]  +  7.0 * x , name='e')
>>> f = e.copy(name='f')
>>> print(repr(f))
sasoptpy.Expression(exp =  -  y[0]  +  7.0 * x , name='f')
```

sasoptpy.Variable.get_dual

`Variable.get_dual()`

Returns the dual value

Returns float

Dual value of the variable

sasoptpy.Variable.get_name

`Variable.get_name()`

Returns the name of the expression

Returns string

Name of the expression

Examples

```
>>> var1 = m.add_variables(name='x')
>>> print(var1.get_name())
x
```

sasoptpy.Variable.get_value

`Variable.get_value()`

Returns the value of the expression after variable values are changed

Returns float

Value of the expression

Examples

```
>>> profit = so.Expression(5 * sales - 3 * material)
>>> m.solve()
>>> print(profit.get_value())
41.0
```

sasoptpy.Variable.mult

`Variable.mult(other)`

Multiplies the `Expression` with a scalar value

Parameters **other** : `Expression` or int

Second expression to be multiplied

Returns `Expression` object

A new `Expression` that represents the multiplication

Notes

- This method is mainly for internal use.
- Multiplying an expression is equivalent to calling this method: 3*(x-y) and (x-y).mult(3) are interchangeable.

sasoptpy.Variable.set_bounds

`Variable.set_bounds(lb=None, ub=None)`

Changes bounds on a variable

Parameters **lb** : float

Lower bound of the variable

ub : float

Upper bound of the variable

Examples

```
>>> x = so.Variable(name='x', lb=0, ub=20)
>>> print(repr(x))
sasoptpy.Variable(name='x', lb=0, ub=20, vartype='CONT')
>>> x.set_bounds(lb=5, ub=15)
>>> print(repr(x))
sasoptpy.Variable(name='x', lb=5, ub=15, vartype='CONT')
```

sasoptpy.Variable.set_init

`Variable.set_init(init=None)`

Changes initial value of a variable

Parameters **init** : float or None

Initial value of the variable

Examples

```
>>> x = so.Variable(name='x')
>>> x.set_init(5)
```

```
>>> y = so.Variable(name='y', init=3)
>>> y.set_init()
```

sasoptpy.Variable.set_name

`Variable.set_name(name)`

Sets the name of the expression

Parameters **name** : string

Name of the expression

Returns string

Name of the expression after resolving conflicts

Examples

```
>>> e = x + 2*y
>>> e.set_name('objective')
```

sasoptpy.Variable.set_permanent

`Variable.set_permanent(name=None)`

Converts a temporary expression into a permanent one

Parameters **name** : string, optional

Name of the expression

Returns string

Name of the expression in the namespace

7.1.4 sasoptpy.VariableGroup

`class sasoptpy.VariableGroup(*argv, name, vartype='CONT', lb=0, ub=inf, init=None)`

Creates a group of `Variable` objects

Parameters **argv** : list, dict, int, `pandas.Index`

Loop index for variable group

name : string, optional

Name (prefix) of the variables

vartype : string, optional

Type of variables, *BIN*, *INT*, or *CONT*

lb : list, dict, `pandas.Series`, optional

Lower bounds of variables

ub : list, dict, `pandas.Series`, optional

Upper bounds of variables

init : float, optional

Initial values of variables

See also:

`sasoptpy.Model.add_variables()`, `sasoptpy.Model.include()`

Notes

- When working with a single model, use the `sasoptpy.Model.add_variables()` method.
- If a variable group object is created, it can be added to a model using the `sasoptpy.Model.include()` method.
- An individual variable inside the group can be accessed using indices.

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z', lb=0, ub=10)
>>> print(repr(z[0, 'a']))
sasoptpy.Variable(name='z_0_a', lb=0, ub=10, vartype='CONT')
```

Examples

```
>>> PERIODS = ['Period1', 'Period2', 'Period3']
>>> production = so.VariableGroup(PERIODS, vartype=so.INT,
                                  name='production', lb=10)
>>> print(production)
Variable Group (production) [
  [Period1: production['Period1']]
  [Period2: production['Period2']]
  [Period3: production['Period3']]
]
```

```
>>> x = so.VariableGroup(4, vartype=so.BIN, name='x')
>>> print(x)
Variable Group (x) [
  [0: x[0]]
  [1: x[1]]
  [2: x[2]]
  [3: x[3]]
]
```

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z')
>>> print(z)
Variable Group (z) [
  [(0, 'a'): z[0, 'a']]
  [(0, 'b'): z[0, 'b']]
  [(0, 'c'): z[0, 'c']]
  [(1, 'a'): z[1, 'a']]
  [(1, 'b'): z[1, 'b']]
  [(1, 'c'): z[1, 'c']]
]
>>> print(repr(z))
sasoptpy.VariableGroup([0, 1], ['a', 'b', 'c'], name='z')
```

Methods

get_name()	Returns the name of the variable group
mult(vector)	Quick multiplication method for the variable groups
set_bounds([lb, ub])	Sets / updates bounds for the given variable
sum(*argv)	Quick sum method for the variable groups

sasoptpy.VariableGroup.get_name

VariableGroup.**get_name**()

Returns the name of the variable group

Returns string

Name of the variable group

Examples

```
>>> var1 = m.add_variables(4, name='x')
>>> print(var1.get_name())
x
```

sasoptpy.VariableGroup.mult

`VariableGroup.mult`(*vector*)

Quick multiplication method for the variable groups

Parameters vector : list, dictionary, `pandas.Series` object, or `pandas.DataFrame` object

Vector to be multiplied with the variable group

Returns *`Expression`* object

An expression that is the product of the variable group with the given vector

Examples

Multiplying with a list

```
>>> x = so.VariableGroup(4, vartype=so.BIN, name='x')
>>> e1 = x.mult([1, 5, 6, 10])
>>> print(e1)
 10.0 * x[3]  +  6.0 * x[2]  +  x[0]  +  5.0 * x[1]
```

Multiplying with a dictionary

```
>>> y = so.VariableGroup([0, 1], ['a', 'b'], name='y', lb=0, ub=10)
>>> dvals = {(0, 'a'): 1, (0, 'b'): 2, (1, 'a'): -1, (1, 'b'): 5}
>>> e2 = y.mult(dvals)
>>> print(e2)
 2.0 * y[0, 'b']  -  y[1, 'a']  +  y[0, 'a']  +  5.0 * y[1, 'b']
```

Multiplying with a pandas.Series object

```
>>> u = so.VariableGroup(['a', 'b', 'c', 'd'], name='u')
>>> ps = pd.Series([0.1, 1.5, -0.2, 0.3], index=['a', 'b', 'c', 'd'])
>>> e3 = u.mult(ps)
>>> print(e3)
 1.5 * u['b']  +  0.1 * u['a']  -  0.2 * u['c']  +  0.3 * u['d']
```

Multiplying with a pandas.DataFrame object

```
>>> data = np.random.rand(3, 3)
>>> df = pd.DataFrame(data, columns=['a', 'b', 'c'])
>>> print(df)
>>> NOTE: Initialized model model1
          a          b          c
```

(continues on next page)

(continued from previous page)

```
0  0.966524  0.237081  0.944630
1  0.821356  0.074753  0.345596
2  0.065229  0.037212  0.136644
>>> y = m.add_variables(3, ['a', 'b', 'c'], name='y')
>>> e = y.mult(df)
>>> print(e)
 0.9665237354418064 * y[0, 'a']  +  0.23708064143289442 * y[0, 'b']  +
0.944629500537536 * y[0, 'c']  +  0.8213562592159828 * y[1, 'a']  +
0.07475256894157478 * y[1, 'b']  +  0.3455957019116668 * y[1, 'c']  +
0.06522945752546017 * y[2, 'a']  +  0.03721153533250843 * y[2, 'b']  +
0.13664422498043194 * y[2, 'c']
```

sasoptpy.VariableGroup.set_bounds

`VariableGroup.`**`set_bounds`**`(`*lb=None*, *ub=None*`)`

Sets / updates bounds for the given variable

Parameters **lb** : Lower bound, optional

ub : Upper bound, optional

Examples

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z', lb=0, ub=10)
>>> print(repr(z[0, 'a']))
sasoptpy.Variable(name='z_0_a', lb=0, ub=10, vartype='CONT')
>>> z.set_bounds(lb=3, ub=5)
>>> print(repr(z[0, 'a']))
sasoptpy.Variable(name='z_0_a', lb=3, ub=5, vartype='CONT')
```

```
>>> u = so.VariableGroup(['a', 'b', 'c', 'd'], name='u')
>>> lb_vals = pd.Series([1, 4, 0, -1], index=['a', 'b', 'c', 'd'])
>>> u.set_bounds(lb=lb_vals)
>>> print(repr(u['b']))
sasoptpy.Variable(name='u_b', lb=4, ub=inf, vartype='CONT')
```

sasoptpy.VariableGroup.sum

`VariableGroup.`**`sum`**`(`**argv*`)`

Quick sum method for the variable groups

Parameters **argv** : Arguments

List of indices for the sum

Returns `Expression` object

Expression that represents the sum of all variables in the group

Examples

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z', lb=0, ub=10)
>>> e1 = z.sum('*', '*')
>>> print(e1)
z[1, 'c']  +  z[1, 'a']  +  z[1, 'b']  +  z[0, 'a']  +  z[0, 'b']  +
z[0, 'c']
>>> e2 = z.sum('*', 'a')
>>> print(e2)
 z[1, 'a']  +  z[0, 'a']
>>> e3 = z.sum('*', ['a', 'b'])
>>> print(e3)
 z[1, 'a']  +  z[0, 'b']  +  z[1, 'b']  +  z[0, 'a']
```

7.1.5 sasoptpy.Constraint

class sasoptpy.**Constraint**(*exp*, *direction=None*, *name=None*, *crange=0*)

Creates a linear or quadratic constraint for optimization models

Constraints should be created by adding logical relations to *Expression* objects.

Parameters **exp** : *Expression*

A logical expression that forms the constraint

direction : string

Direction of the logical expression, 'E' (=), 'L' (<=) or 'G' (>=)

name : string, optional

Name of the constraint object

range : float, optional

Range for ranged constraints

See also:

sasoptpy.Model.add_constraint()

Notes

- A constraint can be generated in multiple ways:

  1. Using the *sasoptpy.Model.add_constraint()* method

```
>>> c1 = m.add_constraint(3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
```

  2. Using the constructor

```
>>> c1 = sasoptpy.Constraint(3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
```

- The same constraint can be included into other models using the *Model.include()* method.

Examples

```
>>> c1 = so.Constraint( 3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
```

```
>>> c2 = so.Constraint( - x + 2 * y - 5, direction='L', name='c2')
sasoptpy.Constraint( -  x  +  2.0 * y  <=  5, name='c2')
```

Methods

add(other[, sign])	Combines two expressions and produces a new one
copy([name])	Returns a copy of the *Expression* object
get_dual()	Returns the dual value
get_name()	Returns the name of the expression
get_value([rhs])	Returns the current value of the constraint
mult(other)	Multiplies the *Expression* with a scalar value
set_block(block_number)	Sets the decomposition block number for a constraint
set_direction(direction)	Changes the direction of a constraint
set_name(name)	Sets the name of the expression
set_permanent([name])	Converts a temporary expression into a permanent one
set_rhs(value)	Changes the RHS of a constraint
update_var_coef(var, value)	Updates the coefficient of a variable inside the constraint

sasoptpy.Constraint.add

`Constraint.`**add**(*other*, *sign=1*)

Combines two expressions and produces a new one

Parameters **other** : float or *Expression* object

Second expression or constant value to be added

sign : int, optional

Sign of the addition, 1 or -1

in_place : boolean, optional

Whether the addition will be performed in place or not

Returns *Expression* object

Notes

- This method is mainly for internal use.
- Adding an expression is equivalent to calling this method: (x-y)+(3*x-2*y) and (x-y).add(3*x-2*y) are interchangeable.

sasoptpy.Constraint.copy

`Constraint.copy(name=None)`
Returns a copy of the *Expression* object

Parameters **name** : string, optional

Name for the copy

Returns *Expression* object

Copy of the object

Examples

```
>>> e = so.Expression(7 * x - y[0], name='e')
>>> print(repr(e))
sasoptpy.Expression(exp =  -  y[0]  +  7.0 * x , name='e')
>>> f = e.copy(name='f')
>>> print(repr(f))
sasoptpy.Expression(exp =  -  y[0]  +  7.0 * x , name='f')
```

sasoptpy.Constraint.get_dual

`Constraint.get_dual()`
Returns the dual value

Returns float

Dual value of the variable

sasoptpy.Constraint.get_name

`Constraint.get_name()`
Returns the name of the expression

Returns string

Name of the expression

Examples

```
>>> var1 = m.add_variables(name='x')
>>> print(var1.get_name())
x
```

sasoptpy.Constraint.get_value

`Constraint.get_value(rhs=False)`
Returns the current value of the constraint

Parameters **rhs** : boolean, optional

Whether constant values (RHS) will be included in the value or not. Default is false

Examples

```
>>> m.solve()
>>> print(c1.get_value())
6.0
>>> print(c1.get_value(rhs=True))
0.0
```

sasoptpy.Constraint.mult

`Constraint.mult(other)`
Multiplies the `Expression` with a scalar value

Parameters other : `Expression` or int

Second expression to be multiplied

Returns `Expression` object

A new `Expression` that represents the multiplication

Notes

- This method is mainly for internal use.
- Multiplying an expression is equivalent to calling this method: 3*(x-y) and (x-y).mult(3) are interchangeable.

sasoptpy.Constraint.set_block

`Constraint.set_block(block_number)`
Sets the decomposition block number for a constraint

Parameters block_number : int

Block number of the constraint

Examples

```
>>> c1 = m.add_constraints((x + 2 * y[i] <= 5 for i in NODES),
                           name='c1')
>>> for i in NODES:
      c1[i].set_block(i)
```

sasoptpy.Constraint.set_direction

`Constraint.set_direction(direction)`
Changes the direction of a constraint

Parameters direction : string

Direction of the constraint, ‘E’, ‘L’, or ‘G’ for equal to, less than or equal to, and greater than or equal to, respectively

Examples

```
>>> c1 = so.Constraint(exp=3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( 3.0 * x  -  5.0 * y  <=  10, name='c1')
>>> c1.set_direction('G')
>>> print(repr(c1))
sasoptpy.Constraint( 3.0 * x  -  5.0 * y  >=  10, name='c1')
```

sasoptpy.Constraint.set_name

`Constraint.`**`set_name`**(*name*)

Sets the name of the expression

Parameters **name** : string

Name of the expression

Returns string

Name of the expression after resolving conflicts

Examples

```
>>> e = x + 2*y
>>> e.set_name('objective')
```

sasoptpy.Constraint.set_permanent

`Constraint.`**`set_permanent`**(*name=None*)

Converts a temporary expression into a permanent one

Parameters **name** : string, optional

Name of the expression

Returns string

Name of the expression in the namespace

sasoptpy.Constraint.set_rhs

`Constraint.`**`set_rhs`**(*value*)

Changes the RHS of a constraint

Parameters **value** : float

New RHS value for the constraint

Examples

```
>>> x = m.add_variable(name='x')
>>> y = m.add_variable(name='y')
>>> c = m.add_constraint(x + 3*y <= 10, name='con_1')
>>> print(c)
x  +  3.0 * y  <=  10
>>> c.set_rhs(5)
>>> print(c)
x  +  3.0 * y  <=  5
```

sasoptpy.Constraint.update_var_coef

Constraint.**update_var_coef**(*var*, *value*)
Updates the coefficient of a variable inside the constraint

Parameters **var** : *Variable* object

Variable to be updated

value : float

Coefficient of the variable in the constraint

See also:

sasoptpy.Model.set_coef()

Examples

```
>>> c1 = so.Constraint(exp=3 * x - 5 * y <= 10, name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  +  3.0 * x  <=  10, name='c1')
>>> c1.update_var_coef(x, -1)
>>> print(repr(c1))
sasoptpy.Constraint( -  5.0 * y  -  x  <=  10, name='c1')
```

7.1.6 sasoptpy.ConstraintGroup

class sasoptpy.**ConstraintGroup**(*argv*, *name*)
Creates a group of *Constraint* objects

Parameters **argv** : GeneratorType object

A Python generator that includes *sasoptpy.Expression* objects

name : string, optional

Name (prefix) of the constraints

See also:

sasoptpy.Model.add_constraints(), *sasoptpy.Model.include()*

Notes

Use *sasoptpy.Model.add_constraints()* when working with a single model.

Examples

```
>>> var_ind = ['a', 'b', 'c', 'd']
>>> u = so.VariableGroup(var_ind, name='u')
>>> t = so.Variable(name='t')
>>> cg = so.ConstraintGroup((u[i] + 2 * t <= 5 for i in var_ind),
                            name='cg')
>>> print(cg)
Constraint Group (cg) [
  [a:  2.0 * t  +  u['a']  <=  5]
  [b:  u['b']  +  2.0 * t  <=  5]
  [c:  2.0 * t  +  u['c']  <=  5]
  [d:  2.0 * t  +  u['d']  <=  5]
]
```

```
>>> z = so.VariableGroup(2, ['a', 'b', 'c'], name='z', lb=0, ub=10)
>>> cg2 = so.ConstraintGroup((2 * z[i, j] + 3 * z[i-1, j] >= 2 for i in
                              [1] for j in ['a', 'b', 'c']), name='cg2')
>>> print(cg2)
Constraint Group (cg2) [
  [(1, 'a'):  3.0 * z[0, 'a']  +  2.0 * z[1, 'a']  >=  2]
  [(1, 'b'):  2.0 * z[1, 'b']  +  3.0 * z[0, 'b']  >=  2]
  [(1, 'c'):  2.0 * z[1, 'c']  +  3.0 * z[0, 'c']  >=  2]
]
```

Methods

get_expressions([rhs])	Returns constraints as a list of expressions
get_name()	Returns the name of the constraint group

sasoptpy.ConstraintGroup.get_expressions

`ConstraintGroup.`**`get_expressions`**(*rhs=False*)

Returns constraints as a list of expressions

Parameters rhs : boolean, optional

Whether to pass the constant part (rhs) of the constraint or not

Returns `pandas.DataFrame`

Returns a DataFrame consisting of constraints as expressions

Examples

```
>>> cg = so.ConstraintGroup((u[i] + 2 * t <= 5 for i in var_ind),
                            name='cg')
>>> ce = cg.get_expressions()
>>> print(ce)
                    cg
c    u['c']  +  2.0 * t
b    u['b']  +  2.0 * t
d    u['d']  +  2.0 * t
```

(continues on next page)

(continued from previous page)

```
a   u['a']  +  2.0 * t
>>> ce_rhs = cg.get_expressions(rhs=True)
>>> print(ce_rhs)
                                    cg
b      u['b']  -  5  +  2.0 * t
c   -  5  +  u['c']  +  2.0 * t
d   -  5  +  u['d']  +  2.0 * t
a   -  5  +  2.0 * t  +  u['a']
```

sasoptpy.ConstraintGroup.get_name

ConstraintGroup.**get_name**()

Returns the name of the constraint group

Returns string

Name of the constraint group

Examples

```
>>> c1 = m.add_constraints((x + y[i] <= 4 for i in indices),
                           name='con1')
>>> print(c1.get_name())
con1
```

7.2 Functions

check_name(name[, ctype])	Checks if a name is in valid and returns a random string if not
dict_to_frame(dictobj[, cols])	Converts dictionaries to DataFrame objects for pretty printing
extract_list_value(tuplist, listname)	Extracts values inside various object types
flatten_frame(df)	Converts a pandas.DataFrame object into pandas.Series
get_namespace()	Prints details of components registered to the global name dictionary
get_obj_by_name(name)	Returns the reference to an object by using the unique name
get_solution_table(*argv[, sort, rhs])	Returns the requested variable names as a DataFrame table
list_length(listobj)	Returns the length of an object if it is a list, tuple or dict
print_model_mps(model)	Prints the MPS representation of the model
quick_sum(argv)	Quick summation function for *Expression* objects
read_frame(df[, cols])	Reads each column in pandas.DataFrame into a list of pandas.Series objects
register_name(name, obj)	Adds the name of a component into the global reference list
reset_globals()	Deletes the references inside the global dictionary and restarts counters
tuple_pack(obj)	Converts a given object to a tuple object

Continued on next page

Table 8 – continued from previous page

tuple_unpack(tp)	Grabs the first element in a tuple, if a tuple is given as argument

7.2.1 sasoptpy.check_name

`sasoptpy.`**`check_name`**(*name*, *ctype=None*)

Checks if a name is in valid and returns a random string if not

Parameters name : str

Name to be checked if unique

Returns str : The given name if valid, a random string otherwise

7.2.2 sasoptpy.dict_to_frame

`sasoptpy.`**`dict_to_frame`**(*dictobj*, *cols=None*)

Converts dictionaries to DataFrame objects for pretty printing

Parameters dictobj : dict

Dictionary to be converted

cols : list, optional

Column names

Returns `DataFrame` object

DataFrame representation of the dictionary

Examples

```
>>> d = {'coal': {'period1': 1, 'period2': 5, 'period3': 7},
>>>      'steel': {'period1': 8, 'period2': 4, 'period3': 3},
>>>      'copper': {'period1': 5, 'period2': 7, 'period3': 9}}
>>> df = so.dict_to_frame(d)
>>> print(df)
        period1  period2  period3
coal          1        5        7
copper        5        7        9
steel         8        4        3
```

7.2.3 sasoptpy.extract_list_value

`sasoptpy.`**`extract_list_value`**(*tuplist*, *listname*)

Extracts values inside various object types

Parameters tuplist : tuple

Key combination to be extracted

listname : dict or list or int or float or DataFrame or Series object

List where the value will be extracted

Returns object

Corresponding value inside listname

7.2.4 sasoptpy.flatten_frame

`sasoptpy.flatten_frame(df)`

Converts a `pandas.DataFrame` object into `pandas.Series`

Parameters df : `pandas.DataFrame` object

Returns `pandas.DataFrame` object

A new DataFrame where indices consist of index and columns names as tuples

Examples

```
>>> price = pd.DataFrame([
>>>     [1, 5, 7],
>>>     [8, 4, 3],
>>>     [5, 7, 9]], columns=['period1', 'period2', 'period3']).\
>>>     set_index([['coal', 'steel', 'copper']])
>>> print('Price data: \n{}'.format(price))
>>> price_f = so.flatten_frame(price)
>>> print('Price data: \n{}'.format(price_f))
Price data:
        period1  period2  period3
coal          1        5        7
steel         8        4        3
copper        5        7        9
Price data:
(coal, period1)      1
(coal, period2)      5
(coal, period3)      7
(steel, period1)     8
(steel, period2)     4
(steel, period3)     3
(copper, period1)    5
(copper, period2)    7
(copper, period3)    9
dtype: int64
```

7.2.5 sasoptpy.get_namespace

`sasoptpy.get_namespace()`

Prints details of components registered to the global name dictionary

The list includes models, variables, constraints and expressions

7.2.6 sasoptpy.get_obj_by_name

`sasoptpy.get_obj_by_name(name)`

Returns the reference to an object by using the unique name

Returns object

Reference to the object that has the name

See also:

reset_globals()

Notes

If there is a conflict in the namespace, you might not get the object you request. Clear the namespace using *reset_globals()* when needed.

Examples

```
>>> m.add_variable(name='var_x', lb=0)
>>> m.add_variables(2, name='var_y', vartype=so.INT)
>>> x = so.get_obj_by_name('var_x')
>>> y = so.get_obj_by_name('var_y')
>>> print(x)
>>> print(y)
>>> m.add_constraint(x + y[0] <= 3, name='con_1')
>>> c1 = so.get_obj_by_name('con_1')
>>> print(c1)
var_x
Variable Group var_y
[(0,): Variable [ var_y_0 | INT ]]
[(1,): Variable [ var_y_1 | INT ]]
var_x  +  var_y_0  <=  3
```

7.2.7 sasoptpy.get_solution_table

sasoptpy.**get_solution_table**(**argv*, *sort=True*, *rhs=False*)

Returns the requested variable names as a DataFrame table

Parameters **sort** : bool, optional

Sort option for the indices

Returns pandas.DataFrame

DataFrame object that holds keys and values

7.2.8 sasoptpy.list_length

sasoptpy.**list_length**(*listobj*)

Returns the length of an object if it is a list, tuple or dict

Parameters **listobj** : Python object

Returns int

Length of the list, tuple or dict, otherwise 1

7.2.9 sasoptpy.print_model_mps

sasoptpy.**print_model_mps**(*model*)

Prints the MPS representation of the model

Parameters **model** : *Model* object

See also:

sasoptpy.Model.to_frame()

Examples

```
>>> m = so.Model(name='print_example', session=s)
>>> x = m.add_variable(lb=1, name='x')
>>> y = m.add_variables(2, name='y', ub=3, vartype=so.INT)
>>> m.add_constraint(x + y.sum('*') <= 9, name='c1')
>>> m.add_constraints((x + y[i] >= 2 for i in [0, 1]), name='c2')
>>> m.set_objective(x+3*y[0], sense=so.MAX, name='obj')
>>> so.print_model_mps(m)
NOTE: Initialized model print_example
     Field1     Field2          Field3 Field4     Field5 Field6 _id_
0      NAME             print_example      0                 0     1
1      ROWS                                                         2
2       MAX        obj                                              3
3         L         c1                                              4
4         G       c2_0                                              5
5         G       c2_1                                              6
6   COLUMNS                                                         7
7                    x             obj      1                       8
8                    x              c1      1                       9
9                    x            c2_0      1                      10
10                   x            c2_1      1                      11
11           MARK0000        'MARKER'          'INTORG'            12
12                 y_0             obj      3                      13
13                 y_0              c1      1                      14
14                 y_0            c2_0      1                      15
15                 y_1              c1      1                      16
16                 y_1            c2_1      1                      17
17           MARK0001        'MARKER'          'INTEND'            18
18      RHS                                                        19
19                 RHS              c1      9                      20
20                 RHS            c2_0      2                      21
21                 RHS            c2_1      2                      22
22   RANGES                                                        23
23   BOUNDS                                                        24
24       LO        BND               x      1                      25
25       UP        BND             y_0      3                      26
26       LO        BND             y_0      0                      27
27       UP        BND             y_1      3                      28
28       LO        BND             y_1      0                      29
29   ENDATA                                 0                 0    30
```

7.2.10 sasoptpy.quick_sum

sasoptpy.**quick_sum**(*argv*)

Quick summation function for *Expression* objects

Returns *Expression* object

Sum of given arguments

Notes

This function is faster for expressions compared to Python's native sum() function.

Examples

```
>>> x = so.VariableGroup(10000, name='x')
>>> y = so.quick_sum(2*x[i] for i in range(10000))
```

7.2.11 sasoptpy.read_frame

sasoptpy.**read_frame**(*df*, *cols=None*)

Reads each column in `pandas.DataFrame` into a list of `pandas.Series` objects

Parameters **df** : `pandas.DataFrame` object

DataFrame to be read

cols : list of strings, optional

Column names to be read. By default, it reads all columns

Returns list

List of `pandas.Series` objects

Examples

```
>>> price = pd.DataFrame([
>>>     [1, 5, 7],
>>>     [8, 4, 3],
>>>     [5, 7, 9]], columns=['period1', 'period2', 'period3']).\
>>>     set_index([['coal', 'steel', 'copper']])
>>> [period2, period3] = so.read_frame(price, ['period2', 'period3'])
>>> print(period2)
coal      5
steel     4
copper    7
Name: period2, dtype: int64
```

7.2.12 sasoptpy.register_name

sasoptpy.**register_name**(*name*, *obj*)

Adds the name of a component into the global reference list

7.2.13 sasoptpy.reset_globals

sasoptpy.**reset_globals**()

Deletes the references inside the global dictionary and restarts counters

See also:

get_namespace()

Examples

```
>>> import sasoptpy as so
>>> m = so.Model(name='my_model')
>>> print(so.get_namespace())
Global namespace:
    Model
           0 my_model <class 'sasoptpy.model.Model'>, sasoptpy.Model(name='my_
↪model', session=None)
    VariableGroup
    ConstraintGroup
    Expression
    Variable
    Constraint
>>> so.reset_globals()
>>> print(so.get_namespace())
Global namespace:
    Model
    VariableGroup
    ConstraintGroup
    Expression
    Variable
    Constraint
```

7.2.14 sasoptpy.tuple_pack

sasoptpy.**tuple_pack**(*obj*)

Converts a given object to a tuple object

If the object is a tuple, the function returns itself, otherwise creates a single dimensional tuple.

Parameters **obj** : Object

Object that is converted to tuple

Returns tuple

Corresponding tuple to the object.

7.2.15 sasoptpy.tuple_unpack

sasoptpy.**tuple_unpack**(*tp*)

Grabs the first element in a tuple, if a tuple is given as argument

Parameters **tp** : tuple

Returns object

The first object inside the tuple.

CHAPTER

EIGHT

EXAMPLES

Examples are provided from SAS/OR documentation.

8.1 Viya (swat) Examples

8.1.1 Food Manufacture 1

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    # Problem data
    OILS = ['veg1', 'veg2', 'oil1', 'oil2', 'oil3']
    PERIODS = range(1, 7)
    cost_data = [
        [110, 120, 130, 110, 115],
        [130, 130, 110, 90, 115],
        [110, 140, 130, 100,  95],
        [120, 110, 120, 120, 125],
        [100, 120, 150, 110, 105],
        [90, 100, 140,  80, 135]]
    cost = pd.DataFrame(cost_data, columns=OILS)
    hardness_data = [8.8, 6.1, 2.0, 4.2, 5.0]
    hardness = {OILS[i]: hardness_data[i] for i in range(len(OILS))}

    revenue_per_ton = 150
    veg_ub = 200
    nonveg_ub = 250
    store_ub = 1000
    storage_cost_per_ton = 5
    hardness_lb = 3
    hardness_ub = 6
    init_storage = 500

    # Problem initialization
    m = so.Model(name='food_manufacture_1', session=cas_conn)
```

(continues on next page)

(continued from previous page)

```
    # Problem definition
    buy = m.add_variables(OILS, PERIODS, lb=0, name='buy')
    use = m.add_variables(OILS, PERIODS, lb=0, name='use')
    manufacture = [use.sum('*', p) for p in PERIODS]
    last_period = len(PERIODS)
    store = m.add_variables(OILS, [0] + list(PERIODS), lb=0, ub=store_ub,
                            name='store')
    for oil in OILS:
        store[oil, 0].set_bounds(lb=init_storage, ub=init_storage)
        store[oil, last_period].set_bounds(lb=init_storage, ub=init_storage)
    VEG = [i for i in OILS if 'veg' in i]
    NONVEG = [i for i in OILS if i not in VEG]
    revenue = so.quick_sum(revenue_per_ton * manufacture[p-1] for p in PERIODS)
    rawcost = so.quick_sum(cost.at[p-1, o] * buy[o, p]
                           for o in OILS for p in PERIODS)
    storagecost = so.quick_sum(storage_cost_per_ton * store[o, p]
                               for o in OILS for p in PERIODS)
    m.set_objective(revenue - rawcost - storagecost, sense=so.MAX,
                    name='profit')

    # Constraints
    m.add_constraints((use.sum(VEG, p) <= veg_ub for p in PERIODS),
                      name='veg_ub')
    m.add_constraints((use.sum(NONVEG, p) <= nonveg_ub for p in PERIODS),
                      name='nonveg_ub')
    m.add_constraints((store[o, p-1] + buy[o, p] == use[o, p] + store[o, p]
                      for o in OILS for p in PERIODS),
                      name='flow_balance')
    m.add_constraints((so.quick_sum(hardness[o]*use[o, p] for o in OILS) >=
                      hardness_lb * manufacture[p-1] for p in PERIODS),
                      name='hardness_ub')
    m.add_constraints((so.quick_sum(hardness[o]*use[o, p] for o in OILS) <=
                      hardness_ub * manufacture[p-1] for p in PERIODS),
                      name='hardness_lb')
    res = m.solve()

    # With other solve options
    m.solve(lp={'algorithm': 'PS'})
    m.solve(lp={'algorithm': 'IP'})
    m.solve(lp={'algorithm': 'NS'})

    if res is not None:
        print(so.get_solution_table(buy, use, store))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.food_manufacture_1 import test

In [2]: test(cas_conn)
NOTE: Initialized model food_manufacture_1.
NOTE: Converting model food_manufacture_1 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMP3EA71LRX␣
↪in caslib CASUSERHDFS(casuser).
```

(continues on next page)

(continued from previous page)

```
NOTE: The table TMP3EA71LRX has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem food_manufacture_1 has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 0 constraints.
NOTE: The LP presolver removed 10 constraint coefficients.
NOTE: The presolved problem has 85 variables, 54 constraints, and 200 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.019986E+06         0
          D 2         54    1.253907E+05         0
          P 2         71    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Converting model food_manufacture_1 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMP64SSN3EJ␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMP64SSN3EJ has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem food_manufacture_1 has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 0 constraints.
NOTE: The LP presolver removed 10 constraint coefficients.
NOTE: The presolved problem has 85 variables, 54 constraints, and 200 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          P 1          1    2.310290E+03         0
          P 2         47    4.276988E+04         0
          P 2         56    8.634295E+04         0
          D 2         70    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Primal Simplex solve time is 0.01 seconds.
NOTE: Converting model food_manufacture_1 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPGGRH98B6␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPGGRH98B6 has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem food_manufacture_1 has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 constraint coefficients.
```

(continues on next page)

(continued from previous page)

```
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 0 constraints.
NOTE: The LP presolver removed 10 constraint coefficients.
NOTE: The presolved problem has 85 variables, 54 constraints, and 200 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 32 threads.
                                           Primal        Bound         Dual
         Iter  Complement Duality Gap      Infeas       Infeas       Infeas   Time
            0  4.2997E+03  1.5010E+01  4.2157E-02  1.4325E-01  4.2366E-01      0
            1  2.7269E+03  4.0309E+00  1.7457E-03  5.9323E-03  2.5977E-01      0
            2  8.0688E+02  7.3876E-01  8.5537E-04  2.9066E-03  6.5752E-02      0
            3  3.8920E+02  3.7862E-01  3.3049E-04  1.1230E-03  8.8814E-03      0
            4  4.1483E+01  3.8035E-02  3.7209E-05  1.2644E-04  6.7674E-04      0
            5  1.2691E+00  1.1121E-03  5.3186E-07  1.8073E-06  2.6917E-05      0
            6  1.2754E-02  1.1177E-05  5.3951E-09  1.8333E-08  2.6964E-07      0
            7  0.0000E+00  8.0023E-08  3.0560E-07  9.4554E-10  8.8666E-07      0
NOTE: The Interior Point solve time is 0.00 seconds.
NOTE: The CROSSOVER option is enabled.
NOTE: The crossover basis contains 11 primal and 3 dual superbasic variables.
                              Objective
         Phase Iteration        Value         Time
          P C          1    8.684742E+02         0
          D C         13    1.703660E+02         0
          D 2         16    1.078426E+05         0
          P 2         17    1.078426E+05         0
          D 2         18    1.078426E+05         0
NOTE: The Crossover time is 0.01 seconds.
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: Converting model food_manufacture_1 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPSQTLC41L␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPSQTLC41L has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem food_manufacture_1 has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 0 constraints.
NOTE: The LP presolver removed 10 constraint coefficients.
NOTE: The presolved problem has 85 variables, 54 constraints, and 200 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 24 rows (44.44%), 51 columns (60.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.
                           Primal          Primal            Dual
         Iteration      Objective  Infeasibility  Infeasibility     Time
                 1   -1.250000E+04   5.000000E+02   4.076000E+03     0.00
                38    5.125000E+04   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
```

(continues on next page)

(continued from previous page)

```
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    4.090791E+05         0
          P 2         42    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Simplex solve time is 0.01 seconds.
            buy          use         store
1    2
oil1 0        -            -   5.000000e+02
oil1 1        0            0   5.000000e+02
oil1 2        0            0   5.000000e+02
oil1 3        0            0   5.000000e+02
oil1 4        0            0   5.000000e+02
oil1 5        0            0   5.000000e+02
oil1 6        0            0   5.000000e+02
oil2 0        -            -   5.000000e+02
oil2 1        0            0   5.000000e+02
oil2 2      250            0   7.500000e+02
oil2 3        0          250   5.000000e+02
oil2 4        0          250   2.500000e+02
oil2 5        0          250   0.000000e+00
oil2 6      750          250   5.000000e+02
oil3 0        -            -   5.000000e+02
oil3 1        0          250   2.500000e+02
oil3 2        0          250   0.000000e+00
oil3 3        0            0  -8.215650e-14
oil3 4        0 -8.21565e-14   0.000000e+00
oil3 5      500            0   5.000000e+02
oil3 6        0            0   5.000000e+02
veg1 0        -            -   5.000000e+02
veg1 1        0      85.1852   4.148148e+02
veg1 2        0      85.1852   3.296296e+02
veg1 3        0      159.259   1.703704e+02
veg1 4        0      11.1111   1.592593e+02
veg1 5        0      159.259   0.000000e+00
veg1 6  659.259      159.259   5.000000e+02
veg2 0        -            -   5.000000e+02
veg2 1        0      114.815   3.851852e+02
veg2 2        0      114.815   2.703704e+02
veg2 3        0      40.7407   2.296296e+02
veg2 4        0      188.889   4.074074e+01
veg2 5        0      40.7407   0.000000e+00
veg2 6  540.741      40.7407   5.000000e+02
Out[2]: 107842.59259259264
```

8.1.2 Food Manufacture 2

Model

```python
import sasoptpy as so
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
def test(cas_conn):

    # Problem data
    OILS = ['veg1', 'veg2', 'oil1', 'oil2', 'oil3']
    PERIODS = range(1, 7)
    cost_data = [
        [110, 120, 130, 110, 115],
        [130, 130, 110, 90, 115],
        [110, 140, 130, 100,  95],
        [120, 110, 120, 120, 125],
        [100, 120, 150, 110, 105],
        [90, 100, 140,  80, 135]]
    cost = pd.DataFrame(cost_data, columns=OILS)
    hardness_data = [8.8, 6.1, 2.0, 4.2, 5.0]
    hardness = {OILS[i]: hardness_data[i] for i in range(len(OILS))}

    revenue_per_ton = 150
    veg_ub = 200
    nonveg_ub = 250
    store_ub = 1000
    storage_cost_per_ton = 5
    hardness_lb = 3
    hardness_ub = 6
    init_storage = 500
    max_num_oils_used = 3
    min_oil_used_threshold = 20

    # Problem initialization
    m = so.Model(name='food_manufacture_2', session=cas_conn)

    # Problem definition
    buy = m.add_variables(OILS, PERIODS, lb=0, name='buy')
    use = m.add_variables(OILS, PERIODS, lb=0, name='use')
    manufacture = [use.sum('*', p) for p in PERIODS]
    last_period = len(PERIODS)
    store = m.add_variables(OILS, [0] + list(PERIODS), lb=0, ub=store_ub,
                            name='store')
    for oil in OILS:
        store[oil, 0].set_bounds(lb=init_storage, ub=init_storage)
        store[oil, last_period].set_bounds(lb=init_storage, ub=init_storage)
    VEG = [i for i in OILS if 'veg' in i]
    NONVEG = [i for i in OILS if i not in VEG]
    revenue = so.quick_sum(revenue_per_ton * manufacture[p-1] for p in PERIODS)
    rawcost = so.quick_sum(cost.at[p-1, o] * buy[o, p]
                           for o in OILS for p in PERIODS)
    storagecost = so.quick_sum(storage_cost_per_ton * store[o, p] for o in OILS
                               for p in PERIODS)
    m.set_objective(revenue - rawcost - storagecost, sense=so.MAX,
                    name='profit')

    # Constraints
    m.add_constraints((use.sum(VEG, p) <= veg_ub for p in PERIODS),
                      name='veg_ub')
    m.add_constraints((use.sum(NONVEG, p) <= nonveg_ub for p in PERIODS),
                      name='nonveg_ub')
    m.add_constraints((store[o, p-1] + buy[o, p] == use[o, p] + store[o, p]
```

(continues on next page)

(continued from previous page)

```
                        for o in OILS for p in PERIODS),
                        name='flow_balance')
    m.add_constraints((so.quick_sum(hardness[o]*use[o, p] for o in OILS) >=
                        hardness_lb * manufacture[p-1] for p in PERIODS),
                        name='hardness_ub')
    m.add_constraints((so.quick_sum(hardness[o]*use[o, p] for o in OILS) <=
                        hardness_ub * manufacture[p-1] for p in PERIODS),
                        name='hardness_lb')

    # Additions to the first problem
    isUsed = m.add_variables(OILS, PERIODS, vartype=so.BIN, name='is_used')
    for p in PERIODS:
        for o in VEG:
            use[o, p].set_bounds(ub=veg_ub)
        for o in NONVEG:
            use[o, p].set_bounds(ub=nonveg_ub)
    m.add_constraints((use[o, p] <= use[o, p]._ub * isUsed[o, p]
                        for o in OILS for p in PERIODS), name='link')
    m.add_constraints((isUsed.sum('*', p) <= max_num_oils_used
                        for p in PERIODS), name='logical1')
    m.add_constraints((use[o, p] >= min_oil_used_threshold * isUsed[o, p]
                        for o in OILS for p in PERIODS), name='logical2')
    m.add_constraints((isUsed[o, p] <= isUsed['oil3', p]
                        for o in ['veg1', 'veg2'] for p in PERIODS),
                        name='logical3')

    res = m.solve()
    if res is not None:
        print(so.get_solution_table(buy, use, store, isUsed))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.food_manufacture_2 import test

In [2]: test(cas_conn)
NOTE: Initialized model food_manufacture_2.
NOTE: Converting model food_manufacture_2 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPFJBYUH31␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPFJBYUH31 has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem food_manufacture_2 has 125 variables (30 binary, 0 integer, 0 free,␣
↪10 fixed).
NOTE: The problem has 132 constraints (66 LE, 30 EQ, 36 GE, 0 range).
NOTE: The problem has 384 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 50 variables and 10 constraints.
NOTE: The MILP presolver removed 66 constraint coefficients.
NOTE: The MILP presolver modified 6 constraint coefficients.
NOTE: The presolved problem has 75 variables, 122 constraints, and 318 constraint␣
↪coefficients.
```

(continues on next page)

(continued from previous page)

```
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      3  77764.2857143         343250   77.34%       0
                0        1      3  77764.2857143         107333   27.55%       0
                0        1      3  77764.2857143         106191   26.77%       0
                0        1      3  77764.2857143         105907   26.57%       0
                0        1      3  77764.2857143         105287   26.14%       0
                0        1      3  77764.2857143         105164   26.05%       0
                0        1      3  77764.2857143         105091   26.00%       0
                0        1      3  77764.2857143         105059   25.98%       0
                0        1      3  77764.2857143         104910   25.87%       0
                0        1      3  77764.2857143         104764   25.77%       0
                0        1      3  77764.2857143         104403   25.52%       0
                0        1      3  77764.2857143         104206   25.37%       0
                0        1      3  77764.2857143         104054   25.27%       0
                0        1      3  77764.2857143         103674   24.99%       0
                0        1      3  77764.2857143         103254   24.69%       0
                0        1      3  77764.2857143         102990   24.49%       0
                0        1      3  77764.2857143         102544   24.17%       0
                0        1      3  77764.2857143         102485   24.12%       0
                0        1      3  77764.2857143         102271   23.96%       0
                0        1      3  77764.2857143         102240   23.94%       0
                0        1      3  77764.2857143         102215   23.92%       0
                0        1      3  77764.2857143         102119   23.85%       0
                0        1      3  77764.2857143         102106   23.84%       0
                0        1      3  77764.2857143         102103   23.84%       0
                0        1      3  77764.2857143         102103   23.84%       0
                0        1      4  87368.5185185         102103   14.43%       0
NOTE: The MILP solver added 34 cuts with 184 cut coefficients at the root.
               60       38      5  99872.2222222         101460    1.57%       1
               92       54      6  99908.3333333         101382    1.45%       1
               95       54      7         100214         101382    1.15%       1
              101       55      8         100279         101356    1.06%       1
              243        0      8         100279         100279    0.00%       1
NOTE: Optimal.
NOTE: Objective = 100278.7037.
                buy          use       store      is_used
1    2
oil1 0            -            -  500.000000            -
oil1 1            0            0  500.000000            0
oil1 2            0            0  500.000000            0
oil1 3            0            0  500.000000            0
oil1 4            0            0  500.000000            0
oil1 5            0            0  500.000000            0
oil1 6            0            0  500.000000            0
oil2 0            -            -  500.000000            -
oil2 1            0           40  460.000000            1
oil2 2            0 -1.76846e-13  460.000000 -1.11022e-15
oil2 3            0            0  460.000000            0
oil2 4            0          230  230.000000            1
oil2 5 -2.84217e-14          230    0.000000            1
oil2 6          730          230  500.000000            1
oil3 0            -            -  500.000000            -
oil3 1            0          210  290.000000            1
oil3 2            0          250   40.000000            1
```

(continues on next page)

(continued from previous page)

```
oil3 3          770          250  560.000000           1
oil3 4            0           20  540.000000           1
oil3 5  1.13687e-13           20  520.000000           1
oil3 6            0           20  500.000000           1
veg1 0            -            -  500.000000           -
veg1 1            0            0  500.000000           0
veg1 2            0      85.1852  414.814815           1
veg1 3            0      85.1852  329.629630           1
veg1 4            0          155  174.629630           1
veg1 5            0          155   19.629630           1
veg1 6       480.37  5.70006e-13  500.000000  3.23892e-15
veg2 0            -            -  500.000000           -
veg2 1            0          200  300.000000           1
veg2 2            0      114.815  185.185185           1
veg2 3  2.84217e-14      114.815   70.370370           1
veg2 4  2.84217e-14            0   70.370370           0
veg2 5            0            0   70.370370           0
veg2 6       629.63          200  500.000000           1
Out[2]: 100278.70370370371
```

8.1.3 Factory Planning 1

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='factory_planning_1', session=cas_conn)

    # Input data
    product_list = ['prod{}'.format(i) for i in range(1, 8)]
    product_data = pd.DataFrame([[10], [6], [8], [4], [11], [9], [3]],
                                columns=['profit']).set_index([product_list])
    demand_data = [
        [500, 1000, 300, 300,  800, 200, 100],
        [600,  500, 200,   0,  400, 300, 150],
        [300,  600,   0,   0,  500, 400, 100],
        [200,  300, 400, 500,  200,   0, 100],
        [0,    100, 500, 100, 1000, 300,   0],
        [500,  500, 100, 300, 1100, 500,  60]]
    demand_data = pd.DataFrame(demand_data, columns=product_list)\
                    .set_index([[i for i in range(1, 7)]])
    machine_types_data = [
        ['grinder', 4],
        ['vdrill', 2],
        ['hdrill', 3],
        ['borer', 1],
        ['planer', 1]]
    machine_types_data = pd.DataFrame(machine_types_data, columns=[
        'machine_type', 'num_machines']).set_index(['machine_type'])
    machine_type_period_data = [
```

(continues on next page)

(continued from previous page)

```
        ['grinder', 1, 1],
        ['hdrill',  2, 2],
        ['borer',   3, 1],
        ['vdrill',  4, 1],
        ['grinder', 5, 1],
        ['vdrill',  5, 1],
        ['planer',  6, 1],
        ['hdrill',  6, 1]]
    machine_type_period_data = pd.DataFrame(machine_type_period_data, columns=[
        'machine_type', 'period', 'num_down'])
    machine_type_product_data = [
        ['grinder', 0.5,  0.7,  0,    0,    0.3,  0.2, 0.5],
        ['vdrill',  0.1,  0.2,  0,    0.3,  0,    0.6, 0],
        ['hdrill',  0.2,  0,    0.8,  0,    0,    0,   0.6],
        ['borer',   0.05, 0.03, 0,    0.07, 0.1,  0,   0.08],
        ['planer',  0,    0,    0.01, 0,    0.05, 0,   0.05]]
    machine_type_product_data = \
        pd.DataFrame(machine_type_product_data, columns=['machine_type'] +
                     product_list).set_index(['machine_type'])
    store_ub = 100
    storage_cost_per_unit = 0.5
    final_storage = 50
    num_hours_per_period = 24 * 2 * 8

    # Problem definition
    PRODUCTS = product_list
    PERIODS = range(1, 7)
    MACHINE_TYPES = machine_types_data.index.values

    num_machine_per_period = pd.DataFrame()
    for i in range(1, 7):
        num_machine_per_period[i] = machine_types_data['num_machines']
    for _, row in machine_type_period_data.iterrows():
        num_machine_per_period.at[row['machine_type'],
                                  row['period']] -= row['num_down']

    make = m.add_variables(PRODUCTS, PERIODS, lb=0, name='make')
    sell = m.add_variables(PRODUCTS, PERIODS, lb=0, ub=demand_data.transpose(),
                           name='sell')

    store = m.add_variables(PRODUCTS, PERIODS, lb=0, ub=store_ub, name='store')
    for p in PRODUCTS:
        store[p, 6].set_bounds(lb=final_storage, ub=final_storage+1)

    storageCost = storage_cost_per_unit * store.sum('*', '*')
    revenue = so.quick_sum(product_data.at[p, 'profit'] * sell[p, t]
                           for p in PRODUCTS for t in PERIODS)
    m.set_objective(revenue-storageCost, sense=so.MAX, name='total_profit')

    production_time = machine_type_product_data
    m.add_constraints((
        so.quick_sum(production_time.at[mc, p] * make[p, t] for p in PRODUCTS)
        <= num_hours_per_period * num_machine_per_period.at[mc, t]
        for mc in MACHINE_TYPES for t in PERIODS), name='machine_hours')
    m.add_constraints(((store[p, t-1] if t-1 in PERIODS else 0) + make[p, t] ==
                       sell[p, t] + store[p, t] for p in PRODUCTS
                       for t in PERIODS),
```

(continues on next page)

(continued from previous page)

```
                      name='flow_balance')

    res = m.solve()
    if res is not None:
        print(so.get_solution_table(make, sell, store))

    print(m.get_solution('Primal'))
    print(m.get_solution('Dual'))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.factory_planning_1 import test

In [2]: test(cas_conn)
NOTE: Initialized model factory_planning_1.
NOTE: Converting model factory_planning_1 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMP1Y1KYKRH␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMP1Y1KYKRH has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem factory_planning_1 has 126 variables (0 free, 6 fixed).
NOTE: The problem has 72 constraints (30 LE, 42 EQ, 0 GE, 0 range).
NOTE: The problem has 281 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 24 variables and 21 constraints.
NOTE: The LP presolver removed 83 constraint coefficients.
NOTE: The presolved problem has 102 variables, 51 constraints, and 198 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.501963E+04         0
          P 2         34    9.371518E+04         0
NOTE: Optimal.
NOTE: Objective = 93715.178571.
NOTE: The Dual Simplex solve time is 0.01 seconds.
                 make          sell  store
1      2
prod1 1   500.000000   500.000000    0.0
prod1 2   700.000000   600.000000  100.0
prod1 3     0.000000   100.000000    0.0
prod1 4   200.000000   200.000000    0.0
prod1 5     0.000000     0.000000    0.0
prod1 6   550.000000   500.000000   50.0
prod2 1   888.571429   888.571429    0.0
prod2 2   600.000000   500.000000  100.0
prod2 3     0.000000   100.000000    0.0
prod2 4   300.000000   300.000000    0.0
prod2 5   100.000000   100.000000    0.0
prod2 6   550.000000   500.000000   50.0
```

(continues on next page)

(continued from previous page)

```
prod3 1   382.500000   300.000000   82.5
prod3 2   117.500000   200.000000    0.0
prod3 3     0.000000     0.000000    0.0
prod3 4   400.000000   400.000000    0.0
prod3 5   600.000000   500.000000  100.0
prod3 6     0.000000    50.000000   50.0
prod4 1   300.000000   300.000000    0.0
prod4 2     0.000000     0.000000    0.0
prod4 3     0.000000     0.000000    0.0
prod4 4   500.000000   500.000000    0.0
prod4 5   100.000000   100.000000    0.0
prod4 6   350.000000   300.000000   50.0
prod5 1   800.000000   800.000000    0.0
prod5 2   500.000000   400.000000  100.0
prod5 3     0.000000   100.000000    0.0
prod5 4   200.000000   200.000000    0.0
prod5 5  1100.000000  1000.000000  100.0
prod5 6     0.000000    50.000000   50.0
prod6 1   200.000000   200.000000    0.0
prod6 2   300.000000   300.000000    0.0
prod6 3   400.000000   400.000000    0.0
prod6 4     0.000000     0.000000    0.0
prod6 5   300.000000   300.000000    0.0
prod6 6   550.000000   500.000000   50.0
prod7 1     0.000000     0.000000    0.0
prod7 2   250.000000   150.000000  100.0
prod7 3     0.000000   100.000000    0.0
prod7 4   100.000000   100.000000    0.0
prod7 5   100.000000     0.000000  100.0
prod7 6     0.000000    50.000000   50.0
Selected Rows from Table PRIMAL

         _OBJ_ID_ _RHS_ID_          _VAR_ _TYPE_  _OBJCOEF_  _LBOUND_  \
0    total_profit      RHS   make_prod1_1      N        0.0       0.0
1    total_profit      RHS   make_prod1_2      N        0.0       0.0
2    total_profit      RHS   make_prod1_3      N        0.0       0.0
3    total_profit      RHS   make_prod1_4      N        0.0       0.0
4    total_profit      RHS   make_prod1_5      N        0.0       0.0
5    total_profit      RHS   make_prod1_6      N        0.0       0.0
6    total_profit      RHS   make_prod2_1      N        0.0       0.0
7    total_profit      RHS   make_prod2_2      N        0.0       0.0
8    total_profit      RHS   make_prod2_3      N        0.0       0.0
9    total_profit      RHS   make_prod2_4      N        0.0       0.0
10   total_profit      RHS   make_prod2_5      N        0.0       0.0
11   total_profit      RHS   make_prod2_6      N        0.0       0.0
12   total_profit      RHS   make_prod3_1      N        0.0       0.0
13   total_profit      RHS   make_prod3_2      N        0.0       0.0
14   total_profit      RHS   make_prod3_3      N        0.0       0.0
15   total_profit      RHS   make_prod3_4      N        0.0       0.0
16   total_profit      RHS   make_prod3_5      N        0.0       0.0
17   total_profit      RHS   make_prod3_6      N        0.0       0.0
18   total_profit      RHS   make_prod4_1      N        0.0       0.0
19   total_profit      RHS   make_prod4_2      N        0.0       0.0
20   total_profit      RHS   make_prod4_3      N        0.0       0.0
21   total_profit      RHS   make_prod4_4      N        0.0       0.0
22   total_profit      RHS   make_prod4_5      N        0.0       0.0
23   total_profit      RHS   make_prod4_6      N        0.0       0.0
```

(continues on next page)

(continued from previous page)

```
24   total_profit      RHS   make_prod5_1      N       0.0       0.0
25   total_profit      RHS   make_prod5_2      N       0.0       0.0
26   total_profit      RHS   make_prod5_3      N       0.0       0.0
27   total_profit      RHS   make_prod5_4      N       0.0       0.0
28   total_profit      RHS   make_prod5_5      N       0.0       0.0
29   total_profit      RHS   make_prod5_6      N       0.0       0.0
..            ...      ...            ...    ...       ...       ...
96   total_profit      RHS  store_prod3_1      D      -0.5       0.0
97   total_profit      RHS  store_prod3_2      D      -0.5       0.0
98   total_profit      RHS  store_prod3_3      D      -0.5       0.0
99   total_profit      RHS  store_prod3_4      D      -0.5       0.0
100  total_profit      RHS  store_prod3_5      D      -0.5       0.0
101  total_profit      RHS  store_prod3_6      D      -0.5      50.0
102  total_profit      RHS  store_prod4_1      D      -0.5       0.0
103  total_profit      RHS  store_prod4_2      D      -0.5       0.0
104  total_profit      RHS  store_prod4_3      D      -0.5       0.0
105  total_profit      RHS  store_prod4_4      D      -0.5       0.0
106  total_profit      RHS  store_prod4_5      D      -0.5       0.0
107  total_profit      RHS  store_prod4_6      D      -0.5      50.0
108  total_profit      RHS  store_prod5_1      D      -0.5       0.0
109  total_profit      RHS  store_prod5_2      D      -0.5       0.0
110  total_profit      RHS  store_prod5_3      D      -0.5       0.0
111  total_profit      RHS  store_prod5_4      D      -0.5       0.0
112  total_profit      RHS  store_prod5_5      D      -0.5       0.0
113  total_profit      RHS  store_prod5_6      D      -0.5      50.0
114  total_profit      RHS  store_prod6_1      D      -0.5       0.0
115  total_profit      RHS  store_prod6_2      D      -0.5       0.0
116  total_profit      RHS  store_prod6_3      D      -0.5       0.0
117  total_profit      RHS  store_prod6_4      D      -0.5       0.0
118  total_profit      RHS  store_prod6_5      D      -0.5       0.0
119  total_profit      RHS  store_prod6_6      D      -0.5      50.0
120  total_profit      RHS  store_prod7_1      D      -0.5       0.0
121  total_profit      RHS  store_prod7_2      D      -0.5       0.0
122  total_profit      RHS  store_prod7_3      D      -0.5       0.0
123  total_profit      RHS  store_prod7_4      D      -0.5       0.0
124  total_profit      RHS  store_prod7_5      D      -0.5       0.0
125  total_profit      RHS  store_prod7_6      D      -0.5      50.0

          _UBOUND_      _VALUE_ _STATUS_   _R_COST_
0    1.797693e+308   500.000000        B  -0.000000
1    1.797693e+308   700.000000        B  -0.000000
2    1.797693e+308     0.000000        B  -0.000000
3    1.797693e+308   200.000000        B  -0.000000
4    1.797693e+308     0.000000        L  -0.000000
5    1.797693e+308   550.000000        B  -0.000000
6    1.797693e+308   888.571429        B  -0.000000
7    1.797693e+308   600.000000        B  -0.000000
8    1.797693e+308     0.000000        L  -0.000000
9    1.797693e+308   300.000000        B  -0.000000
10   1.797693e+308   100.000000        B  -0.000000
11   1.797693e+308   550.000000        B  -0.000000
12   1.797693e+308   382.500000        B  -0.000000
13   1.797693e+308   117.500000        B  -0.000000
14   1.797693e+308     0.000000        L  -0.000000
15   1.797693e+308   400.000000        B  -0.000000
16   1.797693e+308   600.000000        B  -0.000000
17   1.797693e+308     0.000000        B  -0.000000
```

(continues on next page)

(continued from previous page)

```
18   1.797693e+308   300.000000        B  -0.000000
19   1.797693e+308     0.000000        L  -0.000000
20   1.797693e+308     0.000000        L -14.500000
21   1.797693e+308   500.000000        B  -0.000000
22   1.797693e+308   100.000000        B  -0.000000
23   1.797693e+308   350.000000        B  -0.000000
24   1.797693e+308   800.000000        B  -0.000000
25   1.797693e+308   500.000000        B  -0.000000
26   1.797693e+308     0.000000        L  -9.000000
27   1.797693e+308   200.000000        B  -0.000000
28   1.797693e+308  1100.000000        B  -0.000000
29   1.797693e+308     0.000000        L -29.000000
..             ...          ...      ...         ...
96    1.000000e+02    82.500000        B  -0.000000
97    1.000000e+02     0.000000        L  -1.000000
98    1.000000e+02     0.000000        L  -0.500000
99    1.000000e+02     0.000000        L  -0.500000
100   1.000000e+02   100.000000        U   7.500000
101   5.100000e+01    50.000000        L  -8.500000
102   1.000000e+02     0.000000        L  -0.500000
103   1.000000e+02     0.000000        L  -1.000000
104   1.000000e+02    -0.000000        B  -0.000000
105   1.000000e+02     0.000000        L  -0.500000
106   1.000000e+02     0.000000        L  -0.500000
107   5.100000e+01    50.000000        L  -0.500000
108   1.000000e+02     0.000000        L  -3.071429
109   1.000000e+02   100.000000        U  10.500000
110   1.000000e+02     0.000000        L -11.500000
111   1.000000e+02     0.000000        L  -0.500000
112   1.000000e+02   100.000000        U  10.500000
113   5.100000e+01    50.000000        L -11.500000
114   1.000000e+02     0.000000        L  -2.214286
115   1.000000e+02     0.000000        L  -0.500000
116   1.000000e+02     0.000000        L  -0.500000
117   1.000000e+02     0.000000        L  -0.500000
118   1.000000e+02     0.000000        L  -0.500000
119   5.100000e+01    50.000000        L  -0.500000
120   1.000000e+02     0.000000        L  -4.410714
121   1.000000e+02   100.000000        U   2.125000
122   1.000000e+02     0.000000        L  -3.500000
123   1.000000e+02     0.000000        L  -0.500000
124   1.000000e+02   100.000000        U   2.500000
125   5.100000e+01    50.000000        L  -3.500000

[126 rows x 10 columns]
Selected Rows from Table DUAL

        _OBJ_ID_ _RHS_ID_             _ROW_ _TYPE_   _RHS_  _L_RHS_  _U_RHS_  \
0   total_profit      RHS  machine_hours_17      L   768.0      NaN      NaN
1   total_profit      RHS   machine_hours_1      L  1536.0      NaN      NaN
2   total_profit      RHS  machine_hours_25      L   384.0      NaN      NaN
3   total_profit      RHS  machine_hours_22      L   384.0      NaN      NaN
4   total_profit      RHS   machine_hours_7      L   768.0      NaN      NaN
5   total_profit      RHS   machine_hours_5      L  1536.0      NaN      NaN
6   total_profit      RHS  machine_hours_18      L   384.0      NaN      NaN
7   total_profit      RHS  machine_hours_12      L  1152.0      NaN      NaN
8   total_profit      RHS  machine_hours_20      L     0.0      NaN      NaN
```

(continues on next page)

(continued from previous page)

```
9   total_profit      RHS  machine_hours_14      L  1152.0      NaN      NaN
10  total_profit      RHS  machine_hours_23      L   384.0      NaN      NaN
11  total_profit      RHS  machine_hours_16      L  1152.0      NaN      NaN
12  total_profit      RHS   machine_hours_2      L  1536.0      NaN      NaN
13  total_profit      RHS  machine_hours_28      L   384.0      NaN      NaN
14  total_profit      RHS   machine_hours_4      L  1152.0      NaN      NaN
15  total_profit      RHS  machine_hours_11      L   768.0      NaN      NaN
16  total_profit      RHS   machine_hours_0      L  1152.0      NaN      NaN
17  total_profit      RHS  machine_hours_26      L   384.0      NaN      NaN
18  total_profit      RHS  machine_hours_24      L   384.0      NaN      NaN
19  total_profit      RHS   machine_hours_6      L   768.0      NaN      NaN
20  total_profit      RHS   machine_hours_9      L   384.0      NaN      NaN
21  total_profit      RHS  machine_hours_19      L   384.0      NaN      NaN
22  total_profit      RHS  machine_hours_10      L   384.0      NaN      NaN
23  total_profit      RHS   machine_hours_3      L  1536.0      NaN      NaN
24  total_profit      RHS   machine_hours_8      L   768.0      NaN      NaN
25  total_profit      RHS  machine_hours_21      L   384.0      NaN      NaN
26  total_profit      RHS  machine_hours_29      L     0.0      NaN      NaN
27  total_profit      RHS  machine_hours_13      L   384.0      NaN      NaN
28  total_profit      RHS  machine_hours_15      L  1152.0      NaN      NaN
29  total_profit      RHS  machine_hours_27      L   384.0      NaN      NaN
..           ...      ...               ...    ...     ...      ...      ...
42  total_profit      RHS   flow_balance_24      E     0.0      NaN      NaN
43  total_profit      RHS   flow_balance_10      E     0.0      NaN      NaN
44  total_profit      RHS   flow_balance_34      E     0.0      NaN      NaN
45  total_profit      RHS   flow_balance_22      E     0.0      NaN      NaN
46  total_profit      RHS   flow_balance_13      E     0.0      NaN      NaN
47  total_profit      RHS   flow_balance_37      E     0.0      NaN      NaN
48  total_profit      RHS   flow_balance_21      E     0.0      NaN      NaN
49  total_profit      RHS    flow_balance_2      E     0.0      NaN      NaN
50  total_profit      RHS   flow_balance_17      E     0.0      NaN      NaN
51  total_profit      RHS    flow_balance_4      E     0.0      NaN      NaN
52  total_profit      RHS   flow_balance_38      E     0.0      NaN      NaN
53  total_profit      RHS   flow_balance_31      E     0.0      NaN      NaN
54  total_profit      RHS   flow_balance_15      E     0.0      NaN      NaN
55  total_profit      RHS   flow_balance_40      E     0.0      NaN      NaN
56  total_profit      RHS   flow_balance_33      E     0.0      NaN      NaN
57  total_profit      RHS   flow_balance_11      E     0.0      NaN      NaN
58  total_profit      RHS    flow_balance_6      E     0.0      NaN      NaN
59  total_profit      RHS   flow_balance_23      E     0.0      NaN      NaN
60  total_profit      RHS    flow_balance_0      E     0.0      NaN      NaN
61  total_profit      RHS   flow_balance_32      E     0.0      NaN      NaN
62  total_profit      RHS   flow_balance_28      E     0.0      NaN      NaN
63  total_profit      RHS    flow_balance_9      E     0.0      NaN      NaN
64  total_profit      RHS   flow_balance_18      E     0.0      NaN      NaN
65  total_profit      RHS   flow_balance_25      E     0.0      NaN      NaN
66  total_profit      RHS    flow_balance_7      E     0.0      NaN      NaN
67  total_profit      RHS   flow_balance_12      E     0.0      NaN      NaN
68  total_profit      RHS   flow_balance_27      E     0.0      NaN      NaN
69  total_profit      RHS   flow_balance_41      E     0.0      NaN      NaN
70  total_profit      RHS   flow_balance_19      E     0.0      NaN      NaN
71  total_profit      RHS    flow_balance_3      E     0.0      NaN      NaN

      _VALUE_ _STATUS_   _ACTIVITY_
0   -0.000000        B   110.000000
1    0.000000        B  1105.000000
2    0.000000        B    38.675000
```

(continues on next page)

(continued from previous page)

```
3     0.000000        B   128.000000
4     0.000000        B   370.000000
5     0.000000        B   770.000000
6    -0.000000        B   152.657143
7    -0.000000        B   406.000000
8   200.000000        L     0.000000
9    -0.000000        B     0.000000
10    0.000000        B    68.500000
11    0.000000        B   540.000000
12   -0.000000        B    80.000000
13   -0.000000        B    66.000000
14    0.000000        B   510.000000
15    0.000000        B   600.000000
16    8.571429        L  1152.000000
17   -0.000000        B     0.000000
18   -0.000000        B    43.825000
19   -0.000000        B   437.714286
20    0.000000        B   230.000000
21    0.000000        B   123.000000
22    0.000000        B   230.000000
23    0.000000        B   420.000000
24   -0.000000        B   240.000000
25    0.000000        B    82.000000
26  800.000000        L     0.000000
27    0.625000        L   384.000000
28    0.000000        B   420.000000
29   -0.000000        B    19.000000
..         ...      ...          ...
42   -2.571429        U     0.000000
43    0.000000        L     0.000000
44    0.000000        L     0.000000
45    0.000000        L     0.000000
46   -0.500000        U     0.000000
47   -0.375000        U     0.000000
48    0.000000        L     0.000000
49  -10.000000        U     0.000000
50   -8.000000        U     0.000000
51    0.000000        B     0.000000
52   -3.000000        U     0.000000
53    0.000000        L     0.000000
54    0.000000        L     0.000000
55    0.000000        L     0.000000
56    0.000000        B     0.000000
57    0.000000        L     0.000000
58   -6.000000        U     0.000000
59    0.000000        L     0.000000
60   -4.285714        U     0.000000
61    0.000000        L     0.000000
62    0.000000        L     0.000000
63    0.000000        L     0.000000
64    0.000000        U     0.000000
65    0.000000        L     0.000000
66    0.000000        L     0.000000
67    0.000000        U     0.000000
68    0.000000        L     0.000000
69   -3.000000        U     0.000000
70    0.000000        B     0.000000
```

(continues on next page)

(continued from previous page)

```
71    0.000000        L     0.000000

[72 rows x 10 columns]
Out[2]: 93715.17857142858
```

8.1.4 Factory Planning 2

Model

```python
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='factory_planning_2', session=cas_conn)

    # Input data
    product_list = ['prod{}'.format(i) for i in range(1, 8)]
    product_data = pd.DataFrame([10, 6, 8, 4, 11, 9, 3],
                                columns=['profit']).set_index([product_list])

    demand_data = [
        [500, 1000, 300, 300,  800, 200, 100],
        [600,  500, 200,   0,  400, 300, 150],
        [300,  600,   0,   0,  500, 400, 100],
        [200,  300, 400, 500,  200,   0, 100],
        [0,    100, 500, 100, 1000, 300,   0],
        [500,  500, 100, 300, 1100, 500,  60]]
    demand_data = pd.DataFrame(demand_data, columns=product_list)\
                    .set_index([[i for i in range(1, 7)]])
    machine_type_product_data = [
        ['grinder', 0.5,  0.7,  0,    0,    0.3,  0.2, 0.5],
        ['vdrill',  0.1,  0.2,  0,    0.3,  0,    0.6, 0],
        ['hdrill',  0.2,  0,    0.8,  0,    0,    0,   0.6],
        ['borer',   0.05, 0.03, 0,    0.07, 0.1,  0,   0.08],
        ['planer',  0,    0,    0.01, 0,    0.05, 0,   0.05]]
    machine_type_product_data = \
        pd.DataFrame(machine_type_product_data, columns=['machine_type'] +
                     product_list).set_index(['machine_type'])
    machine_types_data = [
        ['grinder', 4, 2],
        ['vdrill',  2, 2],
        ['hdrill',  3, 3],
        ['borer',   1, 1],
        ['planer',  1, 1]]
    machine_types_data = pd.DataFrame(machine_types_data, columns=[
        'machine_type', 'num_machines', 'num_machines_needing_maintenance'])\
        .set_index(['machine_type'])

    store_ub = 100
    storage_cost_per_unit = 0.5
    final_storage = 50
    num_hours_per_period = 24 * 2 * 8
```

(continues on next page)

(continued from previous page)

```
    # Problem definition
    PRODUCTS = product_list
    profit = product_data['profit']
    PERIODS = range(1, 7)
    MACHINE_TYPES = machine_types_data.index.values

    num_machines = machine_types_data['num_machines']

    make = m.add_variables(PRODUCTS, PERIODS, lb=0, name='make')
    sell = m.add_variables(PRODUCTS, PERIODS, lb=0, ub=demand_data.transpose(),
                           name='sell')

    store = m.add_variables(PRODUCTS, PERIODS, lb=0, ub=store_ub, name='store')
    for p in PRODUCTS:
        store[p, 6].set_bounds(lb=final_storage, ub=final_storage)

    storageCost = so.quick_sum(
        storage_cost_per_unit * store[p, t] for p in PRODUCTS for t in PERIODS)
    revenue = so.quick_sum(profit[p] * sell[p, t]
                           for p in PRODUCTS for t in PERIODS)
    m.set_objective(revenue-storageCost, sense=so.MAX, name='total_profit')

    num_machines_needing_maintenance = \
        machine_types_data['num_machines_needing_maintenance']
    numMachinesDown = m.add_variables(MACHINE_TYPES, PERIODS, vartype=so.INT,
                                      lb=0, name='numMachinesDown')

    production_time = machine_type_product_data
    m.add_constraints((
        so.quick_sum(production_time.at[mc, p] * make[p, t] for p in PRODUCTS)
        <= num_hours_per_period *
        (num_machines[mc] - numMachinesDown[mc, t])
        for mc in MACHINE_TYPES for t in PERIODS), name='machine_hours_con')

    m.add_constraints((so.quick_sum(numMachinesDown[mc, t] for t in PERIODS) ==
                       num_machines_needing_maintenance[mc]
                       for mc in MACHINE_TYPES), name='maintenance_con')

    m.add_constraints(((store[p, t-1] if t-1 in PERIODS else 0) + make[p, t] ==
                      sell[p, t] + store[p, t]
                      for p in PRODUCTS for t in PERIODS),
                      name='flow_balance_con')

    res = m.solve()
    if res is not None:
        print(so.get_solution_table(make, sell, store))
        print(so.get_solution_table(numMachinesDown).unstack(level=-1))

    print(m.get_solution_summary())
    print(m.get_problem_summary())

    return m.get_objective_value()
```

Output

```
In [1]: from examples.factory_planning_2 import test

In [2]: test(cas_conn)
NOTE: Initialized model factory_planning_2.
NOTE: Converting model factory_planning_2 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPU80KZ30M␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPU80KZ30M has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem factory_planning_2 has 156 variables (0 binary, 30 integer, 0 free,␣
↪13 fixed).
NOTE: The problem has 77 constraints (30 LE, 47 EQ, 0 GE, 0 range).
NOTE: The problem has 341 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 27 variables and 15 constraints.
NOTE: The MILP presolver removed 63 constraint coefficients.
NOTE: The MILP presolver modified 16 constraint coefficients.
NOTE: The presolved problem has 129 variables, 62 constraints, and 278 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      2  92455.0000000         116455   20.61%       0
                0        1      2  92455.0000000         116455   20.61%       0
                0        1      2  92455.0000000         116155   20.40%       0
                0        1      2  92455.0000000         115453   19.92%       0
                0        1      2  92455.0000000         114236   19.07%       0
                0        1      2  92455.0000000         112243   17.63%       0
                0        1      2  92455.0000000         111415   17.02%       0
                0        1      2  92455.0000000         110318   16.19%       0
                0        1      2  92455.0000000         109641   15.67%       0
                0        1      2  92455.0000000         108974   15.16%       0
                0        1      2  92455.0000000         108891   15.09%       0
                0        1      2  92455.0000000         108858   15.07%       0
                0        1      2  92455.0000000         108855   15.07%       0
                0        0      3         108855         108855    0.00%       0
NOTE: The MILP solver added 30 cuts with 105 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 108855.01206.
                make        sell      store
1     2
prod1 1   500.000000   500.000000    0.000000
prod1 2   600.000000   600.000000    0.000000
prod1 3   399.999335   299.999335  100.000000
prod1 4     0.000000   100.000000    0.000000
prod1 5     0.000000     0.000000    0.000000
prod1 6   550.000000   500.000000   50.000000
prod2 1  1000.000000  1000.000000    0.000000
prod2 2   500.001220   500.000000    0.001220
prod2 3   699.998780   600.000000  100.000000
prod2 4     0.003213   100.003213    0.000000
```

(continues on next page)

(continued from previous page)

```
prod2 5   100.000000   100.000000    0.000000
prod2 6   549.999866   499.999866   50.000000
prod3 1   300.000000   300.000000    0.000000
prod3 2   200.000000   200.000000    0.000000
prod3 3    99.999124     0.000000   99.999124
prod3 4     0.002191   100.001315    0.000000
prod3 5   500.000000   500.000000    0.000000
prod3 6   150.000000   100.000000   50.000000
prod4 1   300.000000   300.000000    0.000000
prod4 2     0.000000     0.000000    0.000000
prod4 3    99.999562     0.000000   99.999562
prod4 4     0.002191   100.001753    0.000000
prod4 5   100.000105   100.000000    0.000105
prod4 6   349.999895   300.000000   50.000000
prod5 1   800.000000   800.000000    0.000000
prod5 2   399.999961   399.999961    0.000000
prod5 3   599.998004   499.998670   99.999335
prod5 4     0.001103   100.000438    0.000000
prod5 5   999.999753   999.999319    0.000435
prod5 6  1149.999565  1100.000000   50.000000
prod6 1   200.000000   200.000000    0.000000
prod6 2   300.000000   300.000000    0.000000
prod6 3   400.000000   400.000000    0.000000
prod6 4     0.000000     0.000000    0.000000
prod6 5   300.000000   300.000000    0.000000
prod6 6   550.000000   500.000000   50.000000
prod7 1   100.000038   100.000000    0.000038
prod7 2   150.000295   150.000000    0.000333
prod7 3   199.998580   100.000000   99.998913
prod7 4     0.001087   100.000000    0.000000
prod7 5     0.000033     0.000000    0.000033
prod7 6   109.999967    60.000000   50.000000
        numMachinesDown numMachinesDown numMachinesDown numMachinesDown  \
2                     1               2               3               4
1
borer               0.0    0.000000e+00        0.000003        0.999996
grinder             0.0    0.000000e+00        0.000000        2.000000
hdrill              1.0    2.000000e+00        0.000000        0.000000
planer              0.0    1.298962e-07        0.000003        0.999996
vdrill              0.0    1.000000e+00        0.000000        0.000000

        numMachinesDown numMachinesDown
2                     5               6
1
borer      7.568570e-07    2.987288e-07
grinder    0.000000e+00    0.000000e+00
hdrill     0.000000e+00    0.000000e+00
planer     9.205018e-07    0.000000e+00
vdrill     1.000000e+00    0.000000e+00
Solution Summary

                                Value
Label
Solver                           MILP
Algorithm              Branch and Cut
Objective Function       total_profit
Solution Status               Optimal
```

(continues on next page)

(continued from previous page)

```
Objective Value          108855.01206

Relative Gap                        0
Absolute Gap                        0
Primal Infeasibility      8.81073E-13
Bound Infeasibility                 0
Integer Infeasibility    4.3817427E-6

Best Bound               108855.01206
Nodes                               1
Solutions Found                     1
Iterations                        283

Presolve Time                    0.02
Solution Time                    1.12
Problem Summary

                                    Value
Label
Problem Name             factory_planning_2
Objective Sense                Maximization
Objective Function             total_profit
RHS                                     RHS

Number of Variables                     156
Bounded Above                             0
Bounded Below                            72
Bounded Above and Below                  71
Free                                      0
Fixed                                    13
Binary                                    0
Integer                                  30

Number of Constraints                    77
LE (<=)                                  30
EQ (=)                                   47
GE (>=)                                   0
Range                                     0

Constraint Coefficients                 341
Out[2]: 108855.01206368084
```

8.1.5 Manpower Planning

Model

```
import sasoptpy as so
import pandas as pd
import math


def test(cas_conn):
    # Input data
    demand_data = pd.DataFrame([
```

(continues on next page)

(continued from previous page)

```
        [0, 2000, 1500, 1000],
        [1, 1000, 1400, 1000],
        [2, 500, 2000, 1500],
        [3, 0, 2500, 2000]
        ], columns=['period', 'unskilled', 'semiskilled', 'skilled'])\
        .set_index(['period'])
    worker_data = pd.DataFrame([
        ['unskilled',   0.25, 0.10, 500, 200, 1500, 50, 500],
        ['semiskilled', 0.20, 0.05, 800, 500, 2000, 50, 400],
        ['skilled',     0.10, 0.05, 500, 500, 3000, 50, 400]
        ], columns=['worker', 'waste_new', 'waste_old', 'recruit_ub',
                    'redundancy_cost', 'overmanning_cost', 'shorttime_ub',
                    'shorttime_cost']).set_index(['worker'])
    retrain_data = pd.DataFrame([
        ['unskilled', 'semiskilled', 200, 400],
        ['semiskilled', 'skilled', math.inf, 500],
        ], columns=['worker1', 'worker2', 'retrain_ub', 'retrain_cost']).\
        set_index(['worker1', 'worker2'])
    downgrade_data = pd.DataFrame([
        ['semiskilled', 'unskilled'],
        ['skilled', 'semiskilled'],
        ['skilled', 'unskilled']
        ], columns=['worker1', 'worker2'])

    semiskill_retrain_frac_ub = 0.25
    downgrade_leave_frac = 0.5
    overmanning_ub = 150
    shorttime_frac = 0.5

    # Sets
    WORKERS = worker_data.index.values
    PERIODS0 = demand_data.index.values
    PERIODS = PERIODS0[1:]
    RETRAIN_PAIRS = [i for i, _ in retrain_data.iterrows()]
    DOWNGRADE_PAIRS = [(row['worker1'], row['worker2'])
                       for _, row in downgrade_data.iterrows()]

    waste_old = worker_data['waste_old']
    waste_new = worker_data['waste_new']
    redundancy_cost = worker_data['redundancy_cost']
    overmanning_cost = worker_data['overmanning_cost']
    shorttime_cost = worker_data['shorttime_cost']
    retrain_cost = retrain_data['retrain_cost'].unstack(level=-1)

    # Initialization
    m = so.Model(name='manpower_planning', session=cas_conn)

    # Variables
    numWorkers = m.add_variables(WORKERS, PERIODS0, name='numWorkers', lb=0)
    demand0 = demand_data.loc[0]
    for w in WORKERS:
        numWorkers[w, 0].set_bounds(lb=demand0[w], ub=demand0[w])
    numRecruits = m.add_variables(WORKERS, PERIODS, name='numRecruits', lb=0)
    worker_ub = worker_data['recruit_ub']
    for w in WORKERS:
        for p in PERIODS:
            numRecruits[w, p].set_bounds(ub=worker_ub[w])
```

(continues on next page)

(continued from previous page)

```
    numRedundant = m.add_variables(WORKERS, PERIODS, name='numRedundant', lb=0)
    numShortTime = m.add_variables(WORKERS, PERIODS, name='numShortTime', lb=0)
    shorttime_ub = worker_data['shorttime_ub']
    for w in WORKERS:
        for p in PERIODS:
            numShortTime.set_bounds(ub=shorttime_ub[w])
    numExcess = m.add_variables(WORKERS, PERIODS, name='numExcess', lb=0)

    retrain_ub = pd.DataFrame()
    for i in PERIODS:
        retrain_ub[i] = retrain_data['retrain_ub']
    numRetrain = m.add_variables(RETRAIN_PAIRS, PERIODS, name='numRetrain',
                                 lb=0, ub=retrain_ub)

    numDowngrade = m.add_variables(DOWNGRADE_PAIRS, PERIODS,
                                   name='numDowngrade', lb=0)
    # Constraints
    m.add_constraints((numWorkers[w, p]
                      - (1-shorttime_frac) * numShortTime[w, p]
                      - numExcess[w, p] == demand_data.loc[p, w]
                      for w in WORKERS for p in PERIODS), name='demand')
    m.add_constraints((numWorkers[w, p] ==
                      (1 - waste_old[w]) * numWorkers[w, p-1]
                      + (1 - waste_new[w]) * numRecruits[w, p]
                      + (1 - waste_old[w]) * numRetrain.sum('*', w, p)
                      + (1 - downgrade_leave_frac) *
                      numDowngrade.sum('*', w, p)
                      - numRetrain.sum(w, '*', p)
                      - numDowngrade.sum(w, '*', p)
                      - numRedundant[w, p]
                      for w in WORKERS for p in PERIODS),
                      name='flow_balance')
    m.add_constraints((numRetrain['semiskilled', 'skilled', p] <=
                      semiskill_retrain_frac_ub * numWorkers['skilled', p]
                      for p in PERIODS), name='semiskill_retrain')
    m.add_constraints((numExcess.sum('*', p) <= overmanning_ub
                      for p in PERIODS), name='overmanning')
    # Objectives
    redundancy = so.Expression(numRedundant.sum('*', '*'), name='redundancy')
    cost = so.Expression(so.quick_sum(redundancy_cost[w] * numRedundant[w, p] +
                                      shorttime_cost[w] * numShortTime[w, p] +
                                      overmanning_cost[w] * numExcess[w, p]
                                      for w in WORKERS for p in PERIODS)
                         + so.quick_sum(
                             retrain_cost.loc[i, j] * numRetrain[i, j, p]
                             for i, j in RETRAIN_PAIRS for p in PERIODS),
                         name='cost')

    m.set_objective(redundancy, sense=so.MIN, name='redundancy_obj')
    res = m.solve()
    if res is not None:
        print(redundancy.get_value())
        print(cost.get_value())
        print(so.get_solution_table(numWorkers, numRecruits, numRedundant,
                                    numShortTime, numExcess))
        print(so.get_solution_table(numRetrain))
        print(so.get_solution_table(numDowngrade))
```

(continues on next page)

(continued from previous page)

```
    m.set_objective(cost, sense=so.MIN, name='cost_obj')
    res = m.solve()
    if res is not None:
        print(redundancy.get_value())
        print(cost.get_value())
        print(so.get_solution_table(numWorkers, numRecruits, numRedundant,
                                    numShortTime, numExcess))
        print(so.get_solution_table(numRetrain))
        print(so.get_solution_table(numDowngrade))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.manpower_planning import test

In [2]: test(cas_conn)
NOTE: Initialized model manpower_planning.
NOTE: Converting model manpower_planning to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPRXOPPKQ_␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPRXOPPKQ_ has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem manpower_planning has 63 variables (0 free, 3 fixed).
NOTE: The problem has 24 constraints (6 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 108 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 21 variables and 9 constraints.
NOTE: The LP presolver removed 21 constraint coefficients.
NOTE: The presolved problem has 42 variables, 15 constraints, and 87 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    5.223600E+02         0
          P 2         13    8.417969E+02         0
NOTE: Optimal.
NOTE: Objective = 841.796875.
NOTE: The Dual Simplex solve time is 0.01 seconds.
841.796875
1462047.6973684211
               numWorkers numRecruits numRedundant numShortTime numExcess
1           2
semiskilled 0  1500.00000           -            -            -         -
semiskilled 1  1442.96875           0            0           50   17.9687
semiskilled 2  2000.00000     682.198            0            0         0
semiskilled 3  2500.00000     645.724            0            0         0
skilled     0  1000.00000           -            -            -         -
skilled     1  1025.00000           0            0           50         0
skilled     2  1525.00000         500            0           50         0
skilled     3  2000.00000         500            0            0         0
```

(continues on next page)

(continued from previous page)

```
unskilled   0  2000.00000          -          -          -        -
unskilled   1  1157.03125          0    442.969         50  132.031
unskilled   2   675.00000          0    166.328         50      150
unskilled   3   175.00000          0      232.5         50      150
                           numRetrain
1           2           3
semiskilled skilled     1  256.250000
semiskilled skilled     2  106.578947
semiskilled skilled     3  106.578947
unskilled   semiskilled 1  200.000000
unskilled   semiskilled 2  200.000000
unskilled   semiskilled 3  200.000000
                           numDowngrade
1           2           3
semiskilled unskilled   1        0.0000
semiskilled unskilled   2        0.0000
semiskilled unskilled   3        0.0000
skilled     semiskilled 1      168.4375
skilled     semiskilled 2        0.0000
skilled     semiskilled 3        0.0000
skilled     unskilled   1        0.0000
skilled     unskilled   2        0.0000
skilled     unskilled   3        0.0000
NOTE: Converting model manpower_planning to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPPKNJM4XN␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPPKNJM4XN has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem manpower_planning has 63 variables (0 free, 3 fixed).
NOTE: The problem has 24 constraints (6 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 108 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 30 variables and 11 constraints.
NOTE: The LP presolver removed 39 constraint coefficients.
NOTE: The presolved problem has 33 variables, 13 constraints, and 69 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    2.143730E+05         0
          D 2          8    4.986773E+05         0
NOTE: Optimal.
NOTE: Objective = 498677.28532.
NOTE: The Dual Simplex solve time is 0.01 seconds.
1423.7188365650968
498677.2853185596
               numWorkers numRecruits numRedundant numShortTime numExcess
1           2
semiskilled 0      1500.0           -            -            -         -
semiskilled 1      1400.0           0            0            0         0
semiskilled 2      2000.0         800            0            0         0
semiskilled 3      2500.0         800            0            0         0
skilled     0      1000.0           -            -            -         -
skilled     1      1000.0     55.5556            0            0         0
```

(continues on next page)

(continued from previous page)

```
skilled     2      1500.0          500           0           0          0
skilled     3      2000.0          500           0           0          0
unskilled   0      2000.0            -           -           -          -
unskilled   1      1000.0            0       812.5           0          0
unskilled   2       500.0            0     257.618           0          0
unskilled   3         0.0            0     353.601           0          0
                               numRetrain
1           2           3
semiskilled skilled     1    0.000000
semiskilled skilled     2  105.263158
semiskilled skilled     3  131.578947
unskilled   semiskilled 1    0.000000
unskilled   semiskilled 2  142.382271
unskilled   semiskilled 3   96.398892
                               numDowngrade
1           2           3
semiskilled unskilled   1          25.0
semiskilled unskilled   2           0.0
semiskilled unskilled   3           0.0
skilled     semiskilled 1           0.0
skilled     semiskilled 2           0.0
skilled     semiskilled 3           0.0
skilled     unskilled   1           0.0
skilled     unskilled   2           0.0
skilled     unskilled   3           0.0
Out[2]: 498677.28531855956
```

8.1.6 Refinery Optimization

Model

```
import sasoptpy as so
import pandas as pd
import numpy as np


def test(cas_conn):

    m = so.Model(name='refinery_optimization', session=cas_conn)

    crude_data = pd.DataFrame([
        ['crude1', 20000],
        ['crude2', 30000]
        ], columns=['crude', 'crude_ub']).set_index(['crude'])

    arc_data = pd.DataFrame([
        ['source', 'crude1', 6],
        ['source', 'crude2', 6],
        ['crude1', 'light_naphtha', 0.1],
        ['crude1', 'medium_naphtha', 0.2],
        ['crude1', 'heavy_naphtha', 0.2],
        ['crude1', 'light_oil', 0.12],
        ['crude1', 'heavy_oil', 0.2],
        ['crude1', 'residuum', 0.13],
```

(continues on next page)

(continued from previous page)

```
        ['crude2', 'light_naphtha', 0.15],
        ['crude2', 'medium_naphtha', 0.25],
        ['crude2', 'heavy_naphtha', 0.18],
        ['crude2', 'light_oil', 0.08],
        ['crude2', 'heavy_oil', 0.19],
        ['crude2', 'residuum', 0.12],
        ['light_naphtha', 'regular_petrol', np.nan],
        ['light_naphtha', 'premium_petrol', np.nan],
        ['medium_naphtha', 'regular_petrol', np.nan],
        ['medium_naphtha', 'premium_petrol', np.nan],
        ['heavy_naphtha', 'regular_petrol', np.nan],
        ['heavy_naphtha', 'premium_petrol', np.nan],
        ['light_naphtha', 'reformed_gasoline', 0.6],
        ['medium_naphtha', 'reformed_gasoline', 0.52],
        ['heavy_naphtha', 'reformed_gasoline', 0.45],
        ['light_oil', 'jet_fuel', np.nan],
        ['light_oil', 'fuel_oil', np.nan],
        ['heavy_oil', 'jet_fuel', np.nan],
        ['heavy_oil', 'fuel_oil', np.nan],
        ['light_oil', 'light_oil_cracked', 2],
        ['light_oil_cracked', 'cracked_oil', 0.68],
        ['light_oil_cracked', 'cracked_gasoline', 0.28],
        ['heavy_oil', 'heavy_oil_cracked', 2],
        ['heavy_oil_cracked', 'cracked_oil', 0.75],
        ['heavy_oil_cracked', 'cracked_gasoline', 0.2],
        ['cracked_oil', 'jet_fuel', np.nan],
        ['cracked_oil', 'fuel_oil', np.nan],
        ['reformed_gasoline', 'regular_petrol', np.nan],
        ['reformed_gasoline', 'premium_petrol', np.nan],
        ['cracked_gasoline', 'regular_petrol', np.nan],
        ['cracked_gasoline', 'premium_petrol', np.nan],
        ['residuum', 'lube_oil', 0.5],
        ['residuum', 'jet_fuel', np.nan],
        ['residuum', 'fuel_oil', np.nan],
        ], columns=['i', 'j', 'multiplier']).set_index(['i', 'j'])

    octane_data = pd.DataFrame([
        ['light_naphtha', 90],
        ['medium_naphtha', 80],
        ['heavy_naphtha', 70],
        ['reformed_gasoline', 115],
        ['cracked_gasoline', 105],
        ], columns=['i', 'octane']).set_index(['i'])

    petrol_data = pd.DataFrame([
        ['regular_petrol', 84],
        ['premium_petrol', 94],
        ], columns=['petrol', 'octane_lb']).set_index(['petrol'])

    vapour_pressure_data = pd.DataFrame([
        ['light_oil', 1.0],
        ['heavy_oil', 0.6],
        ['cracked_oil', 1.5],
        ['residuum', 0.05],
        ], columns=['oil', 'vapour_pressure']).set_index(['oil'])

    fuel_oil_ratio_data = pd.DataFrame([
```

(continues on next page)

(continued from previous page)

```
        ['light_oil', 10],
        ['cracked_oil', 4],
        ['heavy_oil', 3],
        ['residuum', 1],
        ], columns=['oil', 'coefficient']).set_index(['oil'])

    final_product_data = pd.DataFrame([
        ['premium_petrol', 700],
        ['regular_petrol', 600],
        ['jet_fuel', 400],
        ['fuel_oil', 350],
        ['lube_oil', 150],
        ], columns=['product', 'profit']).set_index(['product'])

    vapour_pressure_ub = 1
    crude_total_ub = 45000
    naphtha_ub = 10000
    cracked_oil_ub = 8000
    lube_oil_lb = 500
    lube_oil_ub = 1000
    premium_ratio = 0.40

    ARCS = arc_data.index.tolist()
    arc_mult = arc_data['multiplier'].fillna(1)

    FINAL_PRODUCTS = final_product_data.index.tolist()
    final_product_data['profit'] = final_product_data['profit'] / 100
    profit = final_product_data['profit']

    ARCS = ARCS + [(i, 'sink') for i in FINAL_PRODUCTS]
    flow = m.add_variables(ARCS, name='flow')
    NODES = np.unique([i for j in ARCS for i in j])

    m.set_objective(so.quick_sum(profit[i] * flow[i, 'sink']
                                 for i in FINAL_PRODUCTS
                                 if (i, 'sink') in ARCS),
                    name='totalProfit', sense=so.MAX)

    m.add_constraints((so.quick_sum(flow[a] for a in ARCS if a[0] == n) ==
                      so.quick_sum(arc_mult[a] * flow[a]
                                   for a in ARCS if a[1] == n)
                      for n in NODES if n not in ['source', 'sink']),
                      name='flow_balance')

    CRUDES = crude_data.index.tolist()
    crudeDistilled = m.add_variables(CRUDES, name='crudesDistilled')
    crudeDistilled.set_bounds(ub=crude_data['crude_ub'])
    m.add_constraints((flow[i, j] == crudeDistilled[i]
                      for (i, j) in ARCS if i in CRUDES), name='distillation')

    OILS = ['light_oil', 'heavy_oil']
    CRACKED_OILS = [i+'_cracked' for i in OILS]
    oilCracked = m.add_variables(CRACKED_OILS, name='oilCracked', lb=0)
    m.add_constraints((flow[i, j] == oilCracked[i] for (i, j) in ARCS
                      if i in CRACKED_OILS), name='cracking')

    octane = octane_data['octane']
```

(continues on next page)

(continued from previous page)

```
    PETROLS = petrol_data.index.tolist()
    octane_lb = petrol_data['octane_lb']
    vapour_pressure = vapour_pressure_data['vapour_pressure']

    m.add_constraints((so.quick_sum(octane[a[0]] * arc_mult[a] * flow[a]
                                    for a in ARCS if a[1] == p)
                       >= octane_lb[p] *
                      so.quick_sum(arc_mult[a] * flow[a]
                                   for a in ARCS if a[1] == p)
                      for p in PETROLS), name='blending_petrol')

    m.add_constraint(so.quick_sum(vapour_pressure[a[0]] * arc_mult[a] * flow[a]
                                  for a in ARCS if a[1] == 'jet_fuel') <=
                     vapour_pressure_ub *
                     so.quick_sum(arc_mult[a] * flow[a]
                                  for a in ARCS if a[1] == 'jet_fuel'),
                     name='blending_jet_fuel')

    fuel_oil_coefficient = fuel_oil_ratio_data['coefficient']
    sum_fuel_oil_coefficient = sum(fuel_oil_coefficient)
    m.add_constraints((sum_fuel_oil_coefficient * flow[a] ==
                      fuel_oil_coefficient[a[0]] * flow.sum('*', ['fuel_oil'])
                      for a in ARCS if a[1] == 'fuel_oil'),
                      name='blending_fuel_oil')

    m.add_constraint(crudeDistilled.sum('*') <= crude_total_ub,
                     name='crude_total_ub')

    m.add_constraint(so.quick_sum(flow[a] for a in ARCS
                                  if a[0].find('naphtha') > -1 and
                                  a[1] == 'reformed_gasoline')
                     <= naphtha_ub, name='naphtba_ub')

    m.add_constraint(so.quick_sum(flow[a] for a in ARCS if a[1] ==
                                  'cracked_oil') <=
                     cracked_oil_ub, name='cracked_oil_ub')

    m.add_constraint(flow['lube_oil', 'sink'] == [lube_oil_lb, lube_oil_ub],
                     name='lube_oil_range')

    m.add_constraint(flow.sum('premium_petrol', '*') >= premium_ratio *
                     flow.sum('regular_petrol', '*'), name='premium_ratio')

    print(m.to_frame())

    res = m.solve()
    if res is not None:
        print(so.get_solution_table(crudeDistilled))
        print(so.get_solution_table(oilCracked))
        print(so.get_solution_table(flow))

        octane_sol = []
        for p in PETROLS:
            octane_sol.append(so.quick_sum(octane[a[0]] * arc_mult[a] *
                                           flow[a].get_value() for a in ARCS
                                           if a[1] == p) /
                              sum(arc_mult[a] * flow[a].get_value()
```

(continues on next page)

(continued from previous page)

```
                                    for a in ARCS if a[1] == p))
        octane_sol = pd.Series(octane_sol, name='octane_sol', index=PETROLS)
        print(so.get_solution_table(octane_sol, octane_lb))
        print(so.get_solution_table(vapour_pressure))
        vapour_pressure_sol = sum(vapour_pressure[a[0]] *
                                  arc_mult[a] *
                                  flow[a].get_value() for a in ARCS
                                  if a[1] == 'jet_fuel') /\
            sum(arc_mult[a] * flow[a].get_value() for a in ARCS
                if a[1] == 'jet_fuel')
        print('Vapour_pressure_sol: {:.4f}'.format(vapour_pressure_sol))

        num_fuel_oil_ratio_sol = [arc_mult[a] * flow[a].get_value() /
                                  sum(arc_mult[b] *
                                      flow[b].get_value()
                                      for b in ARCS if b[1] == 'fuel_oil')
                                  for a in ARCS if a[1] == 'fuel_oil']
        num_fuel_oil_ratio_sol = pd.Series(num_fuel_oil_ratio_sol,
                                           name='num_fuel_oil_ratio_sol',
                                           index=[a[0] for a in ARCS
                                                  if a[1] == 'fuel_oil'])
        print(so.get_solution_table(fuel_oil_coefficient,
                                    num_fuel_oil_ratio_sol))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.refinery_optimization import test

In [2]: test(cas_conn)
NOTE: Initialized model refinery_optimization.
NOTE: Converting model refinery_optimization to DataFrame.
     Field1                                  Field2                 Field3  \
0      NAME                                           refinery_optimization
1      ROWS
2       MAX                             totalProfit
3         E                          flow_balance_6
4         E                          flow_balance_7
5         E                          flow_balance_3
6         E                          flow_balance_9
7         E                          flow_balance_4
8         E                          flow_balance_1
9         E                         flow_balance_13
10        E                         flow_balance_16
11        E                         flow_balance_11
12        E                         flow_balance_17
13        E                         flow_balance_14
14        E                          flow_balance_0
15        E                          flow_balance_8
16        E                         flow_balance_12
17        E                         flow_balance_15
18        E                          flow_balance_5
19        E                         flow_balance_10
20        E                          flow_balance_2
```

(continues on next page)

(continued from previous page)

```
21        E                           distillation_1
22        E                           distillation_6
23        E                           distillation_3
24        E                          distillation_10
25        E                           distillation_4
26        E                           distillation_9
27        E                           distillation_8
28        E                           distillation_2
29        E                           distillation_0
..      ...                                      ...                       ...
130         flow_reformed_gasoline_regular_petrol       blending_petrol_0
131                      flow_regular_petrol_sink             totalProfit
132                      flow_regular_petrol_sink           premium_ratio
133                        flow_residuum_fuel_oil     blending_fuel_oil_2
134                        flow_residuum_fuel_oil         flow_balance_17
135                        flow_residuum_fuel_oil     blending_fuel_oil_0
136                        flow_residuum_jet_fuel       blending_jet_fuel
137                        flow_residuum_jet_fuel          flow_balance_8
138                        flow_residuum_lube_oil         flow_balance_12
139                            flow_source_crude1          flow_balance_2
140                            flow_source_crude2          flow_balance_3
141                        crudesDistilled_crude1          distillation_2
142                        crudesDistilled_crude1          distillation_4
143                        crudesDistilled_crude1          crude_total_ub
144                        crudesDistilled_crude1          distillation_0
145                        crudesDistilled_crude2          distillation_9
146                        crudesDistilled_crude2          distillation_8
147                        crudesDistilled_crude2         distillation_11
148                        crudesDistilled_crude2          crude_total_ub
149                   oilCracked_heavy_oil_cracked               cracking_2
150                   oilCracked_light_oil_cracked               cracking_1
151     RHS
152                                           RHS          crude_total_ub
153                                           RHS          cracked_oil_ub
154  RANGES
155                                           rng           lube_oil_range
156  BOUNDS
157      UP                                   BND  crudesDistilled_crude1
158      UP                                   BND  crudesDistilled_crude2
159  ENDATA

    Field4               Field5 Field6 _id_
0        0                          0    1
1                                        2
2                                        3
3                                        4
4                                        5
5                                        6
6                                        7
7                                        8
8                                        9
9                                       10
10                                      11
11                                      12
12                                      13
13                                      14
14                                      15
```

(continues on next page)

(continued from previous page)

```
15                                        16
16                                        17
17                                        18
18                                        19
19                                        20
20                                        21
21                                        22
22                                        23
23                                        24
24                                        25
25                                        26
26                                        27
27                                        28
28                                        29
29                                        30
..      ...                  ...    ...  ...
130     31                               131
131      6       flow_balance_16      1  132
132   -0.4                               133
133     -4        flow_balance_4     -1  134
134      1  blending_fuel_oil_3      17  135
135    -10  blending_fuel_oil_1      -3  136
136  -0.95       flow_balance_17      1  137
137     -1                               138
138   -0.5       flow_balance_17      1  139
139     -6                               140
140     -6                               141
141     -1        distillation_1     -1  142
142     -1        distillation_5     -1  143
143      1        distillation_3     -1  144
144     -1                               145
145     -1        distillation_6     -1  146
146     -1        distillation_7     -1  147
147     -1       distillation_10     -1  148
148      1                               149
149     -1            cracking_3     -1  150
150     -1            cracking_0     -1  151
151                                      152
152  45000            naphtba_ub  10000  153
153   8000        lube_oil_range    500  154
154                                      155
155    500                               156
156                                      157
157  20000                               158
158  30000                               159
159      0                            0  160

[160 rows x 7 columns]
NOTE: Converting model refinery_optimization to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPWD9Z9U8I␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPWD9Z9U8I has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem refinery_optimization has 51 variables (0 free, 0 fixed).
NOTE: The problem has 46 constraints (4 LE, 38 EQ, 3 GE, 1 range).
```

(continues on next page)

(continued from previous page)

```
NOTE: The problem has 158 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 29 variables and 30 constraints.
NOTE: The LP presolver removed 86 constraint coefficients.
NOTE: The presolved problem has 22 variables, 16 constraints, and 72 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    7.181780E+05         0
          P 2         22    2.113651E+05         0
NOTE: Optimal.
NOTE: Objective = 211365.13477.
NOTE: The Dual Simplex solve time is 0.01 seconds.
        crudesDistilled
1
crude1          15000.0
crude2          30000.0
                   oilCracked
1
heavy_oil_cracked      3800.0
light_oil_cracked      4200.0
                                             flow
1                 2
cracked_gasoline  premium_petrol        0.000000
cracked_gasoline  regular_petrol     1936.000000
cracked_oil       fuel_oil              0.000000
cracked_oil       jet_fuel           5706.000000
crude1            heavy_naphtha     15000.000000
crude1            heavy_oil         15000.000000
crude1            light_naphtha     15000.000000
crude1            light_oil         15000.000000
crude1            medium_naphtha    15000.000000
crude1            residuum          15000.000000
crude2            heavy_naphtha     30000.000000
crude2            heavy_oil         30000.000000
crude2            light_naphtha     30000.000000
crude2            light_oil         30000.000000
crude2            medium_naphtha    30000.000000
crude2            residuum          30000.000000
fuel_oil          sink                  0.000000
heavy_naphtha     premium_petrol     1677.804016
heavy_naphtha     reformed_gasoline  5406.861844
heavy_naphtha     regular_petrol     1315.334140
heavy_oil         fuel_oil              0.000000
heavy_oil         heavy_oil_cracked  3800.000000
heavy_oil         jet_fuel           4900.000000
heavy_oil_cracked cracked_gasoline   3800.000000
heavy_oil_cracked cracked_oil        3800.000000
jet_fuel          sink              15156.000000
light_naphtha     premium_petrol     2706.887007
light_naphtha     reformed_gasoline     0.000000
light_naphtha     regular_petrol     3293.112993
light_oil         fuel_oil              0.000000
light_oil         jet_fuel              0.000000
light_oil         light_oil_cracked  4200.000000
```

(continues on next page)

(continued from previous page)

```
light_oil_cracked cracked_gasoline    4200.000000
light_oil_cracked cracked_oil         4200.000000
lube_oil          sink                 500.000000
medium_naphtha    premium_petrol         0.000000
medium_naphtha    reformed_gasoline      0.000000
medium_naphtha    regular_petrol     10500.000000
premium_petrol    sink                6817.778853
reformed_gasoline premium_petrol      2433.087830
reformed_gasoline regular_petrol         0.000000
regular_petrol    sink               17044.447133
residuum          fuel_oil               0.000000
residuum          jet_fuel            4550.000000
residuum          lube_oil            1000.000000
source            crude1             15000.000000
source            crude2             30000.000000
                octane_sol  octane_lb
1
premium_petrol        94.0         94
regular_petrol        84.0         84
             vapour_pressure
1
cracked_oil             1.50
heavy_oil               0.60
light_oil               1.00
residuum                0.05
Vapour_pressure_sol: 0.7737
             coefficient  num_fuel_oil_ratio_sol
1
cracked_oil            4                     NaN
heavy_oil              3                     NaN
light_oil             10                     NaN
residuum               1                     NaN
Out[2]: 211365.134768933
```

8.1.7 Mining Optimization

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='mining_optimization', session=cas_conn)

    mine_data = pd.DataFrame([
        ['mine1', 5, 2, 1.0],
        ['mine2', 4, 2.5, 0.7],
        ['mine3', 4, 1.3, 1.5],
        ['mine4', 5, 3, 0.5],
        ], columns=['mine', 'cost', 'extract_ub', 'quality']).\
        set_index(['mine'])
```

(continues on next page)

(continued from previous page)

```
    year_data = pd.DataFrame([
        [1, 0.9],
        [2, 0.8],
        [3, 1.2],
        [4, 0.6],
        [5, 1.0],
        ], columns=['year', 'quality_required']).set_index(['year'])

    max_num_worked_per_year = 3
    revenue_per_ton = 10
    discount_rate = 0.10

    MINES = mine_data.index.tolist()
    cost = mine_data['cost']
    extract_ub = mine_data['extract_ub']
    quality = mine_data['quality']
    YEARS = year_data.index.tolist()
    quality_required = year_data['quality_required']

    isOpen = m.add_variables(MINES, YEARS, vartype=so.BIN, name='isOpen')
    isWorked = m.add_variables(MINES, YEARS, vartype=so.BIN, name='isWorked')
    extract = m.add_variables(MINES, YEARS, lb=0, name='extract')
    [extract[i, j].set_bounds(ub=extract_ub[i]) for i in MINES for j in YEARS]

    extractedPerYear = {j: extract.sum('*', j) for j in YEARS}
    discount = {j: 1 / (1+discount_rate) ** (j-1) for j in YEARS}

    totalRevenue = revenue_per_ton *\
        so.quick_sum(discount[j] * extractedPerYear[j] for j in YEARS)
    totalCost = so.quick_sum(discount[j] * cost[i] * isOpen[i, j]
                             for i in MINES for j in YEARS)
    m.set_objective(totalRevenue-totalCost, sense=so.MAX, name='totalProfit')

    m.add_constraints((extract[i, j] <= extract[i, j]._ub * isWorked[i, j]
                      for i in MINES for j in YEARS), name='link')

    m.add_constraints((isWorked.sum('*', j) <= max_num_worked_per_year
                      for j in YEARS), name='cardinality')

    m.add_constraints((isWorked[i, j] <= isOpen[i, j] for i in MINES
                      for j in YEARS), name='worked_implies_open')

    m.add_constraints((isOpen[i, j] <= isOpen[i, j-1] for i in MINES
                      for j in YEARS if j != 1), name='continuity')

    m.add_constraints((so.quick_sum(quality[i] * extract[i, j] for i in MINES)
                      == quality_required[j] * extractedPerYear[j]
                      for j in YEARS), name='quality_con')

    res = m.solve()
    if res is not None:
        print(so.get_solution_table(isOpen, isWorked, extract))
        quality_sol = {j: so.quick_sum(quality[i] * extract[i, j].get_value()
                                       for i in MINES)
                       / extractedPerYear[j].get_value() for j in YEARS}
        qs = so.dict_to_frame(quality_sol, ['quality_sol'])
        epy = so.dict_to_frame(extractedPerYear, ['extracted_per_year'])
```

(continues on next page)

(continued from previous page)

```
        print(so.get_solution_table(epy, qs, quality_required))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.mining_optimization import test

In [2]: test(cas_conn)
NOTE: Initialized model mining_optimization.
NOTE: Converting model mining_optimization to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPE75CX22T␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPE75CX22T has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem mining_optimization has 60 variables (40 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 66 constraints (61 LE, 5 EQ, 0 GE, 0 range).
NOTE: The problem has 151 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 8 variables and 8 constraints.
NOTE: The MILP presolver removed 16 constraint coefficients.
NOTE: The MILP presolver modified 11 constraint coefficients.
NOTE: The presolved problem has 52 variables, 58 constraints, and 135 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      7     95.6438817    364.3638322   73.75%       0
                0        1      7     95.6438817    157.7308887   39.36%       0
                0        1      7     95.6438817    153.3061673   37.61%       0
                0        1      7     95.6438817    149.6494350   36.09%       0
                0        1      8    146.8620252    146.8620252    0.00%       0
                0        0      8    146.8620252    146.8620252    0.00%       0
NOTE: The MILP solver added 4 cuts with 20 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 146.86202522.
           isOpen  isWorked   extract
1     2
mine1 1  1.000000  1.000000  2.000000
mine1 2  1.000000  0.000010  0.000020
mine1 3  1.000000  1.000000  1.950000
mine1 4  1.000000  1.000000  0.125000
mine1 5  1.000000  1.000000  2.000000
mine2 1  1.000000  0.000000  0.000000
mine2 2  1.000000  0.999990  2.499976
mine2 3  1.000000  0.000000  0.000000
mine2 4  1.000000  1.000000  2.500000
mine2 5  0.999998  0.999998  2.166667
mine3 1  1.000000  1.000000  1.300000
mine3 2  1.000000  1.000000  1.299997
```

(continues on next page)

(continued from previous page)

```
mine3 3  1.000000  1.000000  1.300000
mine3 4  1.000000  0.000000  0.000000
mine3 5  1.000000  1.000000  1.300000
mine4 1  1.000000  1.000000  2.450000
mine4 2  1.000000  1.000000  2.200013
mine4 3  1.000000  0.000000  0.000000
mine4 4  1.000000  1.000000  3.000000
mine4 5  0.000000  0.000000  0.000000
   extracted_per_year  quality_sol  quality_required
1
1            5.750000          0.9               0.9
2            6.000005          0.8               0.8
3            3.250000          1.2               1.2
4            5.625000          0.6               0.6
5            5.466667          1.0               1.0
Out[2]: 146.86202522326406
```

8.1.8 Farm Planning

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='farm_planning', session=cas_conn)

    # Input Data

    cow_data_raw = []
    for age in range(12):
        if age < 2:
            row = {'age': age,
                   'init_num_cows': 10,
                   'acres_needed': 2/3.0,
                   'annual_loss': 0.05,
                   'bullock_yield': 0,
                   'heifer_yield': 0,
                   'milk_revenue': 0,
                   'grain_req': 0,
                   'sugar_beet_req': 0,
                   'labour_req': 10,
                   'other_costs': 50}
        else:
            row = {'age': age,
                   'init_num_cows': 10,
                   'acres_needed': 1,
                   'annual_loss': 0.02,
                   'bullock_yield': 1.1/2,
                   'heifer_yield': 1.1/2,
                   'milk_revenue': 370,
                   'grain_req': 0.6,
```

(continues on next page)

(continued from previous page)

```
                'sugar_beet_req': 0.7,
                'labour_req': 42,
                'other_costs': 100}
        cow_data_raw.append(row)
    cow_data = pd.DataFrame(cow_data_raw).set_index(['age'])
    grain_data = pd.DataFrame([
        ['group1', 20, 1.1],
        ['group2', 30, 0.9],
        ['group3', 20, 0.8],
        ['group4', 10, 0.65]
        ], columns=['group', 'acres', 'yield']).set_index(['group'])
    num_years = 5
    num_acres = 200
    bullock_revenue = 30
    heifer_revenue = 40
    dairy_cow_selling_age = 12
    dairy_cow_selling_revenue = 120
    max_num_cows = 130
    sugar_beet_yield = 1.5
    grain_cost = 90
    grain_revenue = 75
    grain_labour_req = 4
    grain_other_costs = 15
    sugar_beet_cost = 70
    sugar_beet_revenue = 58
    sugar_beet_labour_req = 14
    sugar_beet_other_costs = 10
    nominal_labour_cost = 4000
    nominal_labour_hours = 5500
    excess_labour_cost = 1.2
    capital_outlay_unit = 200
    num_loan_years = 10
    annual_interest_rate = 0.15
    max_decrease_ratio = 0.50
    max_increase_ratio = 0.75

    # Sets

    AGES = cow_data.index.tolist()
    init_num_cows = cow_data['init_num_cows']
    acres_needed = cow_data['acres_needed']
    annual_loss = cow_data['annual_loss']
    bullock_yield = cow_data['bullock_yield']
    heifer_yield = cow_data['heifer_yield']
    milk_revenue = cow_data['milk_revenue']
    grain_req = cow_data['grain_req']
    sugar_beet_req = cow_data['sugar_beet_req']
    cow_labour_req = cow_data['labour_req']
    cow_other_costs = cow_data['other_costs']

    YEARS = list(range(1, num_years+1))
    YEARS0 = [0] + YEARS

    # Variables

    numCows = m.add_variables(AGES + [dairy_cow_selling_age], YEARS0, lb=0,
                              name='numCows')
```

(continues on next page)

(continued from previous page)

```
    for age in AGES:
        numCows[age, 0].set_bounds(lb=init_num_cows[age],
                                   ub=init_num_cows[age])
    numCows[dairy_cow_selling_age, 0].set_bounds(lb=0, ub=0)

    numBullocksSold = m.add_variables(YEARS, lb=0, name='numBullocksSold')
    numHeifersSold = m.add_variables(YEARS, lb=0, name='numHeifersSold')

    GROUPS = grain_data.index.tolist()
    acres = grain_data['acres']
    grain_yield = grain_data['yield']
    grainAcres = m.add_variables(GROUPS, YEARS, lb=0, name='grainAcres')
    for group in GROUPS:
        for year in YEARS:
            grainAcres[group, year].set_bounds(ub=acres[group])
    grainBought = m.add_variables(YEARS, lb=0, name='grainBought')
    grainSold = m.add_variables(YEARS, lb=0, name='grainSold')

    sugarBeetAcres = m.add_variables(YEARS, lb=0, name='sugarBeetAcres')
    sugarBeetBought = m.add_variables(YEARS, lb=0, name='sugarBeetBought')
    sugarBeetSold = m.add_variables(YEARS, lb=0, name='sugarBeetSold')

    numExcessLabourHours = m.add_variables(YEARS, lb=0,
                                           name='numExcessLabourHours')
    capitalOutlay = m.add_variables(YEARS, lb=0, name='capitalOutlay')

    yearly_loan_payment = (annual_interest_rate * capital_outlay_unit) /\
                          (1 - (1+annual_interest_rate)**(-num_loan_years))

    # Objective function

    revenue = {year:
               bullock_revenue * numBullocksSold[year] +
               heifer_revenue * numHeifersSold[year] +
               dairy_cow_selling_revenue * numCows[dairy_cow_selling_age,
                                                   year] +
               so.quick_sum(milk_revenue[age] * numCows[age, year]
                            for age in AGES) +
               grain_revenue * grainSold[year] +
               sugar_beet_revenue * sugarBeetSold[year]
               for year in YEARS}

    cost = {year:
            grain_cost * grainBought[year] +
            sugar_beet_cost * sugarBeetBought[year] +
            nominal_labour_cost +
            excess_labour_cost * numExcessLabourHours[year] +
            so.quick_sum(cow_other_costs[age] * numCows[age, year]
                         for age in AGES) +
            so.quick_sum(grain_other_costs * grainAcres[group, year]
                         for group in GROUPS) +
            sugar_beet_other_costs * sugarBeetAcres[year] +
            so.quick_sum(yearly_loan_payment * capitalOutlay[y]
                         for y in YEARS if y <= year)
            for year in YEARS}
    profit = {year: revenue[year] - cost[year] for year in YEARS}
```

(continues on next page)

(continued from previous page)

```
    totalProfit = so.quick_sum(profit[year] -
                               yearly_loan_payment * (num_years - 1 + year) *
                               capitalOutlay[year] for year in YEARS)

    m.set_objective(totalProfit, sense=so.MAX, name='totalProfit')

    # Constraints

    m.add_constraints((
        so.quick_sum(acres_needed[age] * numCows[age, year] for age in AGES) +
        so.quick_sum(grainAcres[group, year] for group in GROUPS) +
        sugarBeetAcres[year] <= num_acres
        for year in YEARS), name='num_acres')

    m.add_constraints((
        numCows[age+1, year+1] == (1-annual_loss[age]) * numCows[age, year]
        for age in AGES if age != dairy_cow_selling_age
        for year in YEARS0 if year != num_years), name='aging')

    m.add_constraints((
        numBullocksSold[year] == so.quick_sum(
            bullock_yield[age] * numCows[age, year] for age in AGES)
        for year in YEARS), name='numBullocksSold_def')

    m.add_constraints((
        numCows[0, year] == so.quick_sum(
            heifer_yield[age] * numCows[age, year]
            for age in AGES) - numHeifersSold[year]
        for year in YEARS), name='numHeifersSold_def')

    m.add_constraints((
        so.quick_sum(numCows[age, year] for age in AGES) <= max_num_cows +
        so.quick_sum(capitalOutlay[y] for y in YEARS if y <= year)
        for year in YEARS), name='max_num_cows_def')

    grainGrown = {(group, year): grain_yield[group] * grainAcres[group, year]
                  for group in GROUPS for year in YEARS}
    m.add_constraints((
        so.quick_sum(grain_req[age] * numCows[age, year] for age in AGES) <=
        so.quick_sum(grainGrown[group, year] for group in GROUPS)
        + grainBought[year] - grainSold[year]
        for year in YEARS), name='grain_req_def')

    sugarBeetGrown = {(year): sugar_beet_yield * sugarBeetAcres[year]
                      for year in YEARS}
    m.add_constraints((
        so.quick_sum(sugar_beet_req[age] * numCows[age, year] for age in AGES)
        <=
        sugarBeetGrown[year] + sugarBeetBought[year] - sugarBeetSold[year]
        for year in YEARS), name='sugar_beet_req_def')

    m.add_constraints((
        so.quick_sum(cow_labour_req[age] * numCows[age, year]
                     for age in AGES) +
        so.quick_sum(grain_labour_req * grainAcres[group, year]
                     for group in GROUPS) +
        sugar_beet_labour_req * sugarBeetAcres[year] <=
```

(continues on next page)

(continued from previous page)

```
        nominal_labour_hours + numExcessLabourHours[year]
        for year in YEARS), name='labour_req_def')
    m.add_constraints((profit[year] >= 0 for year in YEARS), name='cash_flow')

    m.add_constraint(so.quick_sum(numCows[age, num_years] for age in AGES
                                  if age >= 2) /
                     sum(init_num_cows[age] for age in AGES if age >= 2) ==
                     [1-max_decrease_ratio, 1+max_increase_ratio],
                     name='final_dairy_cows_range')

    res = m.solve()

    if res is not None:
        print(so.get_solution_table(numCows))
        revenue_df = so.dict_to_frame(revenue, cols=['revenue'])
        cost_df = so.dict_to_frame(cost, cols=['cost'])
        profit_df = so.dict_to_frame(profit, cols=['profit'])
        print(so.get_solution_table(numBullocksSold, numHeifersSold,
                                    capitalOutlay, numExcessLabourHours,
                                    revenue_df, cost_df, profit_df))
        gg_df = so.dict_to_frame(grainGrown, cols=['grainGrown'])
        print(so.get_solution_table(grainAcres, gg_df))
        sbg_df = so.dict_to_frame(sugarBeetGrown, cols=['sugerBeetGrown'])
        print(so.get_solution_table(
            grainBought, grainSold, sugarBeetAcres,
            sbg_df, sugarBeetBought, sugarBeetSold))
        num_acres = so.get_obj_by_name('num_acres')
        na_df = num_acres.get_expressions()
        max_num_cows_con = so.get_obj_by_name('max_num_cows_def')
        mnc_df = max_num_cows_con.get_expressions()
        print(so.get_solution_table(na_df, mnc_df))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.farm_planning import test

In [2]: test(cas_conn)
NOTE: Initialized model farm_planning.
NOTE: Converting model farm_planning to DataFrame.
WARNING: The objective function contains a constant term. An auxiliary variable is␣
↪added.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPE0DPQWXW␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPE0DPQWXW has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem farm_planning has 144 variables (0 free, 14 fixed).
NOTE: The problem has 101 constraints (25 LE, 70 EQ, 5 GE, 1 range).
NOTE: The problem has 780 constraint coefficients.
NOTE: The following columns have no constraint coefficients:
         numCows_12_0
         obj_constant
```

(continues on next page)

(continued from previous page)

```
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 85 variables and 69 constraints.
NOTE: The LP presolver removed 533 constraint coefficients.
NOTE: The presolved problem has 59 variables, 32 constraints, and 247 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 1          1    4.195000E+02         0
          D 2         37    1.744078E+05         0
          D 2         55    1.217192E+05         0
NOTE: Optimal.
NOTE: Objective = 121719.17286.
NOTE: The Dual Simplex solve time is 0.01 seconds.
        numCows
1  2
0  0  10.000000
0  1  22.800000
0  2  11.584427
0  3   0.000000
0  4   0.000000
0  5   0.000000
1  0  10.000000
1  1   9.500000
1  2  21.660000
1  3  11.005205
1  4   0.000000
1  5   0.000000
2  0  10.000000
2  1   9.500000
2  2   9.025000
2  3  20.577000
2  4  10.454945
2  5   0.000000
3  0  10.000000
3  1   9.800000
3  2   9.310000
3  3   8.844500
3  4  20.165460
3  5  10.245846
4  0  10.000000
4  1   9.800000
4  2   9.604000
4  3   9.123800
4  4   8.667610
4  5  19.762151
...          ...
8  0  10.000000
8  1   9.800000
8  2   9.604000
8  3   9.411920
8  4   9.223682
8  5   9.039208
9  0  10.000000
9  1   9.800000
9  2   9.604000
```

(continues on next page)

(continued from previous page)

```
9  3   9.411920
9  4   9.223682
9  5   9.039208
10 0  10.000000
10 1   9.800000
10 2   9.604000
10 3   9.411920
10 4   9.223682
10 5   9.039208
11 0  10.000000
11 1   9.800000
11 2   9.604000
11 3   9.411920
11 4   9.223682
11 5   9.039208
12 0   0.000000
12 1   9.800000
12 2   9.604000
12 3   9.411920
12 4   9.223682
12 5   9.039208

[78 rows x 1 columns]
   numBullocksSold  numHeifersSold  capitalOutlay  numExcessLabourHours  \
1
1        53.735000       30.935000            0.0                   0.0
2        52.341850       40.757423            0.0                   0.0
3        57.435807       57.435807            0.0                   0.0
4        56.964286       56.964286            0.0                   0.0
5        50.853436       50.853436            0.0                   0.0

        revenue          cost        profit
1
1  41494.530000  19588.466667  21906.063333
2  41153.336497  19264.639818  21888.696679
3  45212.490308  19396.435208  25816.055100
4  45860.056078  19034.285714  26825.770363
5  42716.941438  17434.354053  25282.587385
          grainAcres  grainGrown
1      2
group1 1   20.000000   22.000000
group1 2   20.000000   22.000000
group1 3   20.000000   22.000000
group1 4   20.000000   22.000000
group1 5   20.000000   22.000000
group2 1    0.000000    0.000000
group2 2    0.000000    0.000000
group2 3    3.134152    2.820737
group2 4    0.000000    0.000000
group2 5    0.000000    0.000000
group3 1    0.000000    0.000000
group3 2    0.000000    0.000000
group3 3    0.000000    0.000000
group3 4    0.000000    0.000000
group3 5    0.000000    0.000000
group4 1    0.000000    0.000000
group4 2    0.000000    0.000000
```

(continues on next page)

(continued from previous page)

```
group4 3    0.000000    0.000000
group4 4    0.000000    0.000000
group4 5    0.000000    0.000000
   grainBought  grainSold  sugarBeetAcres  sugerBeetGrown  sugarBeetBought  \
1
1    36.620000        0.0       60.766667       91.150000              0.0
2    35.100200        0.0       62.670049       94.005073              0.0
3    37.836507        0.0       65.100304       97.650456              0.0
4    40.142857        0.0       76.428571      114.642857              0.0
5    33.476475        0.0       87.539208      131.308812              0.0

   sugarBeetSold
1
1      22.760000
2      27.388173
3      24.550338
4      42.142857
5      66.586258
   num_acres  max_num_cows_def
1
1      200.0        130.000000
2      200.0        128.411427
3      200.0        115.433945
4      200.0        103.571429
5      200.0         92.460792
Out[2]: 121719.17286133829
```

8.1.9 Economic Planning

Model

```python
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='economic_planning', session=cas_conn)

    industry_data = pd.DataFrame([
        ['coal', 150, 300, 60],
        ['steel', 80, 350, 60],
        ['transport', 100, 280, 30]
        ], columns=['industry', 'init_stocks', 'init_productive_capacity',
                    'demand']).set_index(['industry'])

    production_data = pd.DataFrame([
        ['coal', 0.1, 0.5, 0.4],
        ['steel', 0.1, 0.1, 0.2],
        ['transport', 0.2, 0.1, 0.2],
        ['manpower', 0.6, 0.3, 0.2],
        ], columns=['input', 'coal',
                    'steel', 'transport']).set_index(['input'])
```

(continues on next page)

(continued from previous page)

```
    productive_capacity_data = pd.DataFrame([
        ['coal', 0.0, 0.7, 0.9],
        ['steel', 0.1, 0.1, 0.2],
        ['transport', 0.2, 0.1, 0.2],
        ['manpower', 0.4, 0.2, 0.1],
        ], columns=['input', 'coal',
                    'steel', 'transport']).set_index(['input'])

    manpower_capacity = 470
    num_years = 5

    YEARS = list(range(1, num_years+1))
    YEARS0 = [0] + list(YEARS)
    INDUSTRIES = industry_data.index.tolist()
    [init_stocks, init_productive_capacity, demand] = so.read_frame(
        industry_data)
    # INPUTS = production_data.index.tolist()
    production_coeff = so.flatten_frame(production_data)
    productive_capacity_coeff = so.flatten_frame(productive_capacity_data)

    static_production = m.add_variables(INDUSTRIES, lb=0,
                                        name='static_production')
    m.set_objective(0, sense=so.MIN, name='Zero')
    m.add_constraints((static_production[i] == demand[i] +
                       so.quick_sum(
                           production_coeff[i, j] * static_production[j]
                           for j in INDUSTRIES) for i in INDUSTRIES),
                      name='static_con')

    m.solve()
    print(so.get_solution_table(static_production))

    final_demand = so.get_solution_table(
        static_production)['static_production']
    # Alternative way
    # final_demand = {}
    # for i in INDUSTRIES:
    #     final_demand[i] = static_production.get_value()

    production = m.add_variables(INDUSTRIES, range(0, num_years+2), lb=0,
                                 name='production')
    stock = m.add_variables(INDUSTRIES, range(0, num_years+2), lb=0,
                            name='stock')
    extra_capacity = m.add_variables(INDUSTRIES, range(1, num_years+3), lb=0,
                                     name='extra_capacity')
    productive_capacity = {}
    for i in INDUSTRIES:
        for year in range(1, num_years+2):
            productive_capacity[i, year] = init_productive_capacity[i] +\
                so.quick_sum(extra_capacity[i, y] for y in range(2, year+1))
    for i in INDUSTRIES:
        production[i, 0].set_bounds(ub=0)
        stock[i, 0].set_bounds(lb=init_stocks[i], ub=init_stocks[i])

    total_productive_capacity = sum(productive_capacity[i, num_years]
                                    for i in INDUSTRIES)
    total_production = so.quick_sum(production[i, year] for i in INDUSTRIES
```

(continues on next page)

(continued from previous page)

```
                                for year in [4, 5])
    total_manpower = so.quick_sum(production_coeff['manpower', i] *
                                  production[i, year+1] +
                                  productive_capacity_coeff['manpower', i] *
                                  extra_capacity[i, year+2]
                                  for i in INDUSTRIES for year in YEARS)

    continuity_con = m.add_constraints((
        stock[i, year] + production[i, year] ==
        (demand[i] if year in YEARS else 0) +
        so.quick_sum(production_coeff[i, j] * production[j, year+1] +
                     productive_capacity_coeff[i, j] *
                     extra_capacity[j, year+2] for j in INDUSTRIES) +
        stock[i, year+1]
        for i in INDUSTRIES for year in YEARS0), name='continuity_con')

    manpower_con = m.add_constraints((
        so.quick_sum(production_coeff['manpower', j] * production[j, year] +
                     productive_capacity_coeff['manpower', j] *
                     extra_capacity[j, year+1]
                     for j in INDUSTRIES)
        <= manpower_capacity for year in range(1, num_years+2)),
        name='manpower_con')

    capacity_con = m.add_constraints((production[i, year] <=
                                      productive_capacity[i, year]
                                      for i in INDUSTRIES
                                      for year in range(1, num_years+2)),
                                     name='capacity_con')

    for i in INDUSTRIES:
        production[i, num_years+1].set_bounds(lb=final_demand[i])

    for i in INDUSTRIES:
        for year in [num_years+1, num_years+2]:
            extra_capacity[i, year].set_bounds(ub=0)

    problem1 = so.Model(name='Problem1', session=cas_conn)
    problem1.include(production, stock, extra_capacity,
                     continuity_con, manpower_con, capacity_con)
    problem1.set_objective(total_productive_capacity, sense=so.MAX,
                           name='total_productive_capacity')
    problem1.solve()
    productive_capacity_fr = so.dict_to_frame(productive_capacity,
                                              cols=['productive_capacity'])
    print(so.get_solution_table(production, stock, extra_capacity,
                                productive_capacity_fr))
    print(so.get_solution_table(manpower_con.get_expressions()))

    # Problem 2

    problem2 = so.Model(name='Problem2', session=cas_conn)
    problem2.include(problem1)
    problem2.set_objective(total_production, name='total_production',
                           sense=so.MAX)
    for i in INDUSTRIES:
        for year in YEARS:
```

(continues on next page)

(continued from previous page)

```
            continuity_con[i, year].set_rhs(0)
    problem2.solve()
    print(so.get_solution_table(production, stock, extra_capacity,
                                productive_capacity))
    print(so.get_solution_table(manpower_con.get_expressions()))

    # Problem 3

    problem3 = so.Model(name='Problem3', session=cas_conn)
    problem3.include(production, stock, extra_capacity, continuity_con,
                     capacity_con)
    problem3.set_objective(total_manpower, sense=so.MAX, name='total_manpower')
    for i in INDUSTRIES:
        for year in YEARS:
            continuity_con[i, year].set_rhs(demand[i])
    problem3.solve()
    print(so.get_solution_table(production, stock, extra_capacity,
                                productive_capacity))
    print(so.get_solution_table(manpower_con.get_expressions()))

    return problem3.get_objective_value()
```

Output

```
In [1]: from examples.economic_planning import test

In [2]: test(cas_conn)
NOTE: Initialized model economic_planning.
NOTE: Converting model economic_planning to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPC9BX6XV9␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPC9BX6XV9 has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem economic_planning has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 9 constraint coefficients.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 0.
           static_production
1
coal              166.396761
steel             105.668016
transport          92.307692
NOTE: Initialized model Problem1.
NOTE: Converting model Problem1 to DataFrame.
WARNING: The objective function contains a constant term. An auxiliary variable is␣
↪added.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPAI1ZFSCS␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPAI1ZFSCS has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
```

(continues on next page)

(continued from previous page)

```
NOTE: Added action set 'optimization'.
NOTE: The problem Problem1 has 64 variables (0 free, 13 fixed).
NOTE: The problem has 42 constraints (24 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 255 constraint coefficients.
NOTE: The following columns have no constraint coefficients:
         extra_capacity_coal_1
         extra_capacity_transport_1
         extra_capacity_steel_1
         obj_constant
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 22 variables and 7 constraints.
NOTE: The LP presolver removed 64 constraint coefficients.
NOTE: The presolved problem has 42 variables, 35 constraints, and 191 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.360782E+04         0
          P 2         39    2.141875E+03         0
NOTE: Optimal.
NOTE: Objective = 2141.8751967.
NOTE: The Dual Simplex solve time is 0.01 seconds.
            production        stock extra_capacity productive_capacity
1         2
coal      0          0          150              -                   -
coal      1    260.403            0              0                 300
coal      2    293.406            0              0                 300
coal      3        300            0              0                 300
coal      4    17.9487      148.448        189.203             489.203
coal      5    166.397            0        1022.67             1511.88
coal      6    166.397 -7.10543e-15              0             1511.88
coal      7          -            -              0                   -
steel     0          0           80              -                   -
steel     1    135.342      12.2811              0                 350
steel     2     181.66            0              0                 350
steel     3     193.09            0              0                 350
steel     4    105.668            0              0                 350
steel     5    105.668            0              0                 350
steel     6    105.668 -7.10543e-15              0                 350
steel     7          -            -              0                   -
transport 0          0          100              -                   -
transport 1    140.722      6.24084              0                 280
transport 2     200.58            0              0                 280
transport 3    267.152            0              0                 280
transport 4    92.3077            0              0                 280
transport 5    92.3077            0              0                 280
transport 6    92.3077  7.10543e-15              0                 280
transport 7          -            -              0                   -
   manpower_con
1
1    224.988515
2    270.657715
3    367.038878
4    470.000000
5    150.000000
6    150.000000
```

(continues on next page)

(continued from previous page)

```
NOTE: Initialized model Problem2.
NOTE: Converting model Problem2 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPCVW8PYNI␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPCVW8PYNI has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem Problem2 has 64 variables (0 free, 13 fixed).
NOTE: The problem has 42 constraints (24 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 255 constraint coefficients.
NOTE: The following columns have no constraint coefficients:
         extra_capacity_coal_1
         extra_capacity_transport_1
         extra_capacity_steel_1
         obj_constant
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 22 variables and 7 constraints.
NOTE: The LP presolver removed 64 constraint coefficients.
NOTE: The presolved problem has 42 variables, 35 constraints, and 191 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.413902E+03         0
          P 2         46    2.618579E+03         0
NOTE: Optimal.
NOTE: Objective = 2618.5791147.
NOTE: The Dual Simplex solve time is 0.01 seconds.
            production    stock extra_capacity     dict
1         2
coal      0          0      150              -        -
coal      1    184.818  31.6285              0      300
coal      2    430.505  16.3725        130.505  430.505
coal      3    430.505        0              0  430.505
coal      4    430.505        0              0  430.505
coal      5    430.505        0              0  430.505
coal      6    166.397  324.108              0  430.505
coal      7          -        -              0        -
steel     0          0       80              -        -
steel     1    86.7295  11.5323              0      350
steel     2    155.337        0              0      350
steel     3    182.867        0              0      350
steel     4    359.402        0        9.40227  359.402
steel     5    359.402  176.535              0  359.402
steel     6    105.668  490.269              0  359.402
steel     7          -        -              0        -
transport 0          0      100              -        -
transport 1    141.312        0              0      280
transport 2    198.388        0              0      280
transport 3    225.918        0              0      280
transport 4    519.383        0        239.383  519.383
transport 5    519.383  293.465              0  519.383
transport 6    92.3077   750.54              0  519.383
transport 7          -        -              0        -
   manpower_con
```

(continues on next page)

(continued from previous page)

```
1
1    217.374162
2    344.581624
3    384.165212
4    470.000000
5    470.000000
6    150.000000
NOTE: Initialized model Problem3.
NOTE: Converting model Problem3 to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMP31DVR5RC␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMP31DVR5RC has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem Problem3 has 63 variables (0 free, 12 fixed).
NOTE: The problem has 36 constraints (18 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 219 constraint coefficients.
NOTE: The following columns have no constraint coefficients:
         extra_capacity_coal_1
         extra_capacity_transport_1
         extra_capacity_steel_1
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 18 variables and 3 constraints.
NOTE: The LP presolver removed 31 constraint coefficients.
NOTE: The presolved problem has 45 variables, 33 constraints, and 188 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    4.013232E+04         0
          P 2         50    2.450027E+03         0
NOTE: Optimal.
NOTE: Objective = 2450.0266228.
NOTE: The Dual Simplex solve time is 0.01 seconds.
            production    stock extra_capacity     dict
1         2
coal      0          0      150              -        -
coal      1    251.793        0              0      300
coal      2    316.015        0        16.0152  316.015
coal      3    319.832        0         3.8168  319.832
coal      4     366.35        0        46.5177   366.35
coal      5     859.36        0         493.01   859.36
coal      6     859.36  460.208              0   859.36
coal      7          -        -              0        -
steel     0          0       80              -        -
steel     1    134.795   11.028              0      350
steel     2    175.041        0              0      350
steel     3    224.064        0              0      350
steel     4    223.136        0              0      350
steel     5    220.044        0              0      350
steel     6        350        0              0      350
steel     7          -        -              0        -
transport 0          0      100              -        -
transport 1    143.559  4.24723              0      280
transport 2    181.676        0              0      280
```

(continues on next page)

(continued from previous page)

```
transport 3       280       0             0     280
transport 4   279.072       0             0     280
transport 5    275.98       0             0     280
transport 6   195.539       0             0     280
transport 7         -       -             0       -
   manpower_con
1
1    226.631832
2    279.983537
3    333.725517
4    539.769130
5    636.824849
6    659.723590
Out[2]: 2450.026622821299
```

8.1.10 Decentralization

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='decentralization', session=cas_conn)

    DEPTS = ['A', 'B', 'C', 'D', 'E']
    CITIES = ['Bristol', 'Brighton', 'London']

    benefit_data = pd.DataFrame([
        ['Bristol', 10, 15, 10, 20, 5],
        ['Brighton', 10, 20, 15, 15, 15]],
        columns=['city'] + DEPTS).set_index('city')

    comm_data = pd.DataFrame([
        ['A', 'B', 0.0],
        ['A', 'C', 1.0],
        ['A', 'D', 1.5],
        ['A', 'E', 0.0],
        ['B', 'C', 1.4],
        ['B', 'D', 1.2],
        ['B', 'E', 0.0],
        ['C', 'D', 0.0],
        ['C', 'E', 2.0],
        ['D', 'E', 0.7]], columns=['i', 'j', 'comm']).set_index(['i', 'j'])

    cost_data = pd.DataFrame([
        ['Bristol', 'Bristol', 5],
        ['Bristol', 'Brighton', 14],
        ['Bristol', 'London', 13],
        ['Brighton', 'Brighton', 5],
        ['Brighton', 'London', 9],
        ['London', 'London', 10]], columns=['i', 'j', 'cost']).set_index(
```

(continues on next page)

(continued from previous page)

```
        ['i', 'j'])

max_num_depts = 3

benefit = {}
for city in CITIES:
    for dept in DEPTS:
        try:
            benefit[dept, city] = benefit_data.ix[city, dept]
        except:
            benefit[dept, city] = 0

comm = {}
for row in comm_data.iterrows():
    (i, j) = row[0]
    comm[i, j] = row[1]['comm']
    comm[j, i] = comm[i, j]

cost = {}
for row in cost_data.iterrows():
    (i, j) = row[0]
    cost[i, j] = row[1]['cost']
    cost[j, i] = cost[i, j]

assign = m.add_variables(DEPTS, CITIES, vartype=so.BIN, name='assign')
IJKL = [(i, j, k, l)
        for i in DEPTS for j in CITIES for k in DEPTS for l in CITIES
        if i < k]
product = m.add_variables(IJKL, vartype=so.BIN)

totalBenefit = so.quick_sum(benefit[i, j] * assign[i, j]
                            for i in DEPTS for j in CITIES)

totalCost = so.quick_sum(comm[i, k] * cost[j, l] * product[i, j, k, l]
                         for (i, j, k, l) in IJKL)

m.set_objective(totalBenefit-totalCost, name='netBenefit', sense=so.MAX)

m.add_constraints((so.quick_sum(assign[dept, city] for city in CITIES)
                  == 1 for dept in DEPTS), name='assign_dept')

m.add_constraints((so.quick_sum(assign[dept, city] for dept in DEPTS)
                  <= max_num_depts for city in CITIES), name='cardinality')

product_def1 = m.add_constraints((assign[i, j] + assign[k, l] - 1
                                 <= product[i, j, k, l]
                                 for (i, j, k, l) in IJKL),
                                 name='product_def1')

product_def2 = m.add_constraints((product[i, j, k, l] <= assign[i, j]
                                  for (i, j, k, l) in IJKL),
                                 name='product_def2')

product_def3 = m.add_constraints((product[i, j, k, l] <= assign[k, l]
                                  for (i, j, k, l) in IJKL),
                                 name='product_def3')
```

(continues on next page)

(continued from previous page)

```
    m.solve()
    print(m.get_problem_summary())

    m.drop_constraints(product_def1)
    m.drop_constraints(product_def2)
    m.drop_constraints(product_def3)

    m.add_constraints((
        so.quick_sum(product[i, j, k, l]
                     for j in CITIES if (i, j, k, l) in IJKL) == assign[k, l]
        for i in DEPTS for k in DEPTS for l in CITIES if i < k),
        name='product_def4')

    m.add_constraints((
        so.quick_sum(product[i, j, k, l]
                     for l in CITIES if (i, j, k, l) in IJKL) == assign[i, j]
        for k in DEPTS for i in DEPTS for j in CITIES if i < k),
        name='product_def4')

    m.solve()
    print(m.get_problem_summary())
    totalBenefit.set_name('totalBenefit')
    totalCost.set_name('totalCost')
    print(so.get_solution_table(totalBenefit, totalCost))
    print(so.get_solution_table(assign).unstack(level=-1))

    return m.get_objective_value()
```

Output

```
In [1]: from examples.decentralization import test

In [2]: test(cas_conn)
NOTE: Initialized model decentralization.
NOTE: Converting model decentralization to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPZZ43MUFB␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPZZ43MUFB has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem decentralization has 105 variables (105 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 278 constraints (183 LE, 5 EQ, 90 GE, 0 range).
NOTE: The problem has 660 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 120 constraints.
NOTE: The MILP presolver removed 120 constraint coefficients.
NOTE: The MILP presolver added 120 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 105 variables, 158 constraints, and 540 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
```

(continues on next page)

(continued from previous page)

```
NOTE: The Branch and Cut algorithm is using up to 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      3     14.9000000     67.5000000   77.93%       0
                0        1      3     14.9000000     52.7500000   71.75%       0
                0        1      3     14.9000000     52.2500000   71.48%       0
                0        1      3     14.9000000     49.6250000   69.97%       0
                0        1      3     14.9000000     42.7500000   65.15%       0
                0        1      3     14.9000000     39.2500000   62.04%       0
                0        1      3     14.9000000     35.7142857   58.28%       0
                0        1      3     14.9000000     34.4000000   56.69%       0
                0        1      3     14.9000000     33.9444444   56.10%       0
                0        0      3     14.9000000     14.9000000    0.00%       0
NOTE: The MILP solver added 37 cuts with 187 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 14.9.
Problem Summary

                                    Value
Label
Problem Name              decentralization
Objective Sense               Maximization
Objective Function              netBenefit
RHS                                    RHS

Number of Variables                    105
Bounded Above                            0
Bounded Below                            0
Bounded Above and Below                105
Free                                     0
Fixed                                    0
Binary                                 105
Integer                                  0

Number of Constraints                  278
LE (<=)                                183
EQ (=)                                   5
GE (>=)                                 90
Range                                    0

Constraint Coefficients                660
NOTE: Converting model decentralization to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPI6KHWKHO␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPI6KHWKHO has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem decentralization has 105 variables (105 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 68 constraints (3 LE, 65 EQ, 0 GE, 0 range).
NOTE: The problem has 270 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 105 variables, 68 constraints, and 270 constraint␣
↪coefficients.
```

(continues on next page)

(continued from previous page)

```
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      2    -12.4000000    135.0000000  109.19%       0
                0        1      2    -12.4000000     30.0000000  141.33%       0
                0        1      3     14.9000000     14.9000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 14.9.
Problem Summary

                                    Value
Label
Problem Name              decentralization
Objective Sense               Maximization
Objective Function              netBenefit
RHS                                    RHS

Number of Variables                    105
Bounded Above                            0
Bounded Below                            0
Bounded Above and Below                105
Free                                     0
Fixed                                    0
Binary                                 105
Integer                                  0

Number of Constraints                   68
LE (<=)                                  3
EQ (=)                                  65
GE (>=)                                  0
Range                                    0

Constraint Coefficients                270
   totalBenefit  totalCost
1
           80.0       65.1
    assign  assign assign
2 Brighton Bristol London
1
A      0.0     1.0    0.0
B      1.0     0.0    0.0
C      1.0     0.0    0.0
D      0.0     1.0    0.0
E      1.0     0.0    0.0
Out[2]: 14.9
```

8.1.11 Optimal Wedding

SAS Blog: https://blogs.sas.com/content/operations/2014/11/10/do-you-have-an-uncle-louie-optimal-wedding-seat-assignments/

Model

```python
import sasoptpy as so
import math


def test(cas_conn, num_guests=10, max_table_size=3, max_tables=None):

    m = so.Model("wedding", session=cas_conn)

    # Check max. tables
    if max_tables is None:
        max_tables = math.ceil(num_guests/max_table_size)

    # Sets
    guests = range(1, num_guests+1)
    tables = range(1, max_tables+1)
    guest_pairs = [[i, j] for i in guests for j in range(i+1, num_guests+1)]

    # Variables
    x = m.add_variables(guests, tables, vartype=so.BIN, name="x")
    unhappy = m.add_variables(tables, name="unhappy", lb=0)

    # Objective
    m.set_objective(unhappy.sum('*'), sense=so.MIN, name="obj")

    # Constraints
    m.add_constraints((x.sum(g, '*') == 1 for g in guests), name="assigncon")
    m.add_constraints((x.sum('*', t) <= max_table_size for t in tables),
                      name="tablesizecon")
    m.add_constraints((unhappy[t] >= abs(g-h)*(x[g, t] + x[h, t] - 1)
                       for t in tables for [g, h] in guest_pairs),
                      name="measurecon")

    # Solve
    res = m.solve(milp={'decomp': {'method': 'set'}, 'presolver': 'none'})

    if res is not None:

        print(so.get_solution_table(x))

        # Print assignments
        for t in tables:
            print('Table {} : [ '.format(t), end='')
            for g in guests:
                if x[g, t].get_value() == 1:
                    print('{} '.format(g), end='')
            print(']')

    return m.get_objective_value()
```

Output

```
In [1]: from examples.sas_optimal_wedding import test

In [2]: test(cas_conn)
```

(continues on next page)

(continued from previous page)

```
NOTE: Initialized model wedding.
NOTE: Converting model wedding to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPHPNRORC3␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPHPNRORC3 has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem wedding has 44 variables (40 binary, 0 integer, 0 free, 0 fixed).
NOTE: The problem has 194 constraints (4 LE, 10 EQ, 180 GE, 0 range).
NOTE: The problem has 620 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in the distributed computing␣
↪environment in single-machine mode.
NOTE: The DECOMP method value SET is applied.
NOTE: The number of block threads has been reduced to 4 threads.
NOTE: The problem has a decomposable structure with 4 blocks. The largest block␣
↪covers 23.71% of the constraints in the problem.
NOTE: The decomposition subproblems cover 44 (100%) variables and 184 (94.85%)␣
↪constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.
         Iter         Best       Master         Best       LP       IP  CPU Real
                     Bound    Objective      Integer      Gap      Gap Time Time
            .       0.0000      12.0000      12.0000 1.20e+01 1.20e+01    0    0
           10       0.0000      12.0000       6.0000 1.20e+01 6.00e+00    6    7
           16       6.0000       6.0000       6.0000    0.00%    0.00%   10   13
            Node   Active    Sols         Best         Best      Gap    CPU   Real
                                       Integer        Bound            Time   Time
               0        0       7       6.0000       6.0000    0.00%     10     13
NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 13.60 seconds.
NOTE: Optimal.
NOTE: Objective = 6.
        x
1  2
1  1  0.0
1  2  0.0
1  3  1.0
1  4  0.0
2  1  0.0
2  2  0.0
2  3  1.0
2  4  0.0
3  1  0.0
3  2  0.0
3  3  1.0
3  4  0.0
4  1  0.0
4  2  0.0
4  3  0.0
4  4  1.0
5  1  0.0
5  2  0.0
```

(continues on next page)

(continued from previous page)

```
5   3   0.0
5   4   1.0
6   1   0.0
6   2   0.0
6   3   0.0
6   4   1.0
7   1   0.0
7   2   1.0
7   3   0.0
7   4   0.0
8   1   0.0
8   2   1.0
8   3   0.0
8   4   0.0
9   1   1.0
9   2   0.0
9   3   0.0
9   4   0.0
10  1   1.0
10  2   0.0
10  3   0.0
10  4   0.0
Table 1 : [ 9 10 ]
Table 2 : [ 7 8 ]
Table 3 : [ 1 2 3 ]
Table 4 : [ 4 5 6 ]
Out[2]: 6.0
```

8.1.12 Kidney Exchange

SAS Blog: https://blogs.sas.com/content/operations/2015/02/06/the-kidney-exchange-problem/

Model

```
import sasoptpy as so
import random


def test(cas_conn):
    # Data generation
    n = 80
    p = 0.02

    random.seed(1)

    ARCS = {}
    for i in range(0, n):
        for j in range(0, n):
            if random.random() < p:
                ARCS[i, j] = random.random()

    max_length = 10
```

(continues on next page)

(continued from previous page)

```
    # Model
    model = so.Model("kidney_exchange", session=cas_conn)

    # Sets
    NODES = set().union(*ARCS.keys())
    MATCHINGS = range(1, int(len(NODES)/2)+1)

    # Variables
    UseNode = model.add_variables(NODES, MATCHINGS, vartype=so.BIN,
                                  name="usenode")
    UseArc = model.add_variables(ARCS, MATCHINGS, vartype=so.BIN,
                                 name="usearc")
    Slack = model.add_variables(NODES, vartype=so.BIN, name="slack")

    print('Setting objective...')

    # Objective
    model.set_objective(so.quick_sum((ARCS[i, j] * UseArc[i, j, m]
                                      for [i, j] in ARCS for m in MATCHINGS)),
                        name="total_weight", sense=so.MAX)

    print('Adding constraints...')
    # Constraints
    Node_Packing = model.add_constraints((UseNode.sum(i, '*') + Slack[i] == 1
                                          for i in NODES), name="node_packing")
    Donate = model.add_constraints((UseArc.sum(i, '*', m) == UseNode[i, m]
                                    for i in NODES
                                    for m in MATCHINGS), name="donate")
    Receive = model.add_constraints((UseArc.sum('*', j, m) == UseNode[j, m]
                                     for j in NODES
                                     for m in MATCHINGS), name="receive")
    Cardinality = model.add_constraints((UseArc.sum('*', '*', m) <= max_length
                                         for m in MATCHINGS),
                                        name="cardinality")

    # Solve
    model.solve(milp={'maxtime': 300})

    # Define decomposition blocks
    for i in NODES:
        for m in MATCHINGS:
            Donate[i, m].set_block(m-1)
            Receive[i, m].set_block(m-1)
    for m in MATCHINGS:
        Cardinality[m].set_block(m-1)

    model.solve(milp={'maxtime': 300, 'presolver': 'basic',
                      'decomp': {'method': 'user'}})

    return model.get_objective_value()
```

Output

```
In [1]: from examples.sas_kidney_exchange import test
```

(continues on next page)

(continued from previous page)

```
In [2]: test(cas_conn)
NOTE: Initialized model kidney_exchange.
Setting objective...
Adding constraints...
NOTE: Converting model kidney_exchange to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPZT5TBUPR␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPZT5TBUPR has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem kidney_exchange has 8133 variables (8133 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 5967 constraints (38 LE, 5929 EQ, 0 GE, 0 range).
NOTE: The problem has 24245 constraint coefficients.
NOTE: The remaining solution time after solver initialization is 299.87 seconds.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 6216 variables and 5356 constraints.
NOTE: The MILP presolver removed 17276 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 1917 variables, 611 constraints, and 6969 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      3      4.5256201   2194.9865951   99.79%       2
                0        1      3      4.5256201     18.3085704   75.28%       3
NOTE: The MILP solver's symmetry detection found 774 orbits. The largest orbit␣
↪contains 15 variables.
               12       10      4     14.7200815     18.3085704   19.60%       4
               22       15      5     17.1113590     18.3085704    6.54%       4
               60        2      7     17.1113590     18.0210902    5.05%      11
               97        0      8     17.1113590     17.1113590    0.00%      12
NOTE: Optimal.
NOTE: Objective = 17.111358985.
NOTE: Cloud Analytic Services made the uploaded file available as table BLOCKSTABLE␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table BLOCKSTABLE has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Converting model kidney_exchange to DataFrame.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPSPP8B14F␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPSPP8B14F has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
NOTE: The problem kidney_exchange has 8133 variables (8133 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 5967 constraints (38 LE, 5929 EQ, 0 GE, 0 range).
NOTE: The problem has 24245 constraint coefficients.
NOTE: The remaining solution time after solver initialization is 299.88 seconds.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 2685 variables and 1925 constraints.
NOTE: The MILP presolver removed 8005 constraint coefficients.
```

(continues on next page)

(continued from previous page)

```
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5448 variables, 4042 constraints, and 16240␣
↪constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in the distributed computing␣
↪environment in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 38 blocks. The largest block␣
↪covers 2.598% of the constraints in the problem.
NOTE: The decomposition subproblems cover 5396 (99.05%) variables and 3990 (98.71%)␣
↪constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.
         Iter         Best       Master         Best       LP        IP  CPU  Real
                     Bound    Objective      Integer      Gap       Gap Time  Time
            .     283.4155      10.6475      10.6475   96.24%    96.24%    3     2
            1     283.4155      10.6475      10.6475   96.24%    96.24%    4     4
            2     229.9494      10.6475      10.6475   95.37%    95.37%    6     5
            3     220.2164      14.8383      14.8383   93.26%    93.26%    8     7
            4     209.0289      14.8383      14.8383   92.90%    92.90%   10     8
            5     150.2142      14.8383      14.8383   90.12%    90.12%   11     9
            7     150.2142      17.1114      17.1114   88.61%    88.61%   14    11
            8     149.2953      17.1114      17.1114   88.54%    88.54%   15    13
            9     125.0268      17.1114      17.1114   86.31%    86.31%   17    14
            .     125.0268      17.1114      17.1114   86.31%    86.31%   18    15
           10      25.4420      17.1114      17.1114   32.74%    32.74%   20    16
           11      19.0558      17.1114      17.1114   10.20%    10.20%   26    18
           12      17.1114      17.1114      17.1114    0.00%     0.00%   33    20
            Node  Active    Sols         Best         Best      Gap    CPU    Real
                                      Integer        Bound             Time   Time
               0       0       6      17.1114      17.1114    0.00%      33     20
NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 20.15 seconds.
NOTE: Optimal.
NOTE: Objective = 17.111358985.
Out[2]: 17.111358984870215
```

8.2 SAS (saspy) Examples

8.2.1 Decentralization (saspy)

Model

```python
import sasoptpy as so
import pandas as pd


def test(cas_conn):

    m = so.Model(name='decentralization', session=cas_conn)

    DEPTS = ['A', 'B', 'C', 'D', 'E']
```

(continues on next page)

(continued from previous page)

```
CITIES = ['Bristol', 'Brighton', 'London']

benefit_data = pd.DataFrame([
    ['Bristol', 10, 15, 10, 20, 5],
    ['Brighton', 10, 20, 15, 15, 15]],
    columns=['city'] + DEPTS).set_index('city')

comm_data = pd.DataFrame([
    ['A', 'B', 0.0],
    ['A', 'C', 1.0],
    ['A', 'D', 1.5],
    ['A', 'E', 0.0],
    ['B', 'C', 1.4],
    ['B', 'D', 1.2],
    ['B', 'E', 0.0],
    ['C', 'D', 0.0],
    ['C', 'E', 2.0],
    ['D', 'E', 0.7]], columns=['i', 'j', 'comm']).set_index(['i', 'j'])

cost_data = pd.DataFrame([
    ['Bristol', 'Bristol', 5],
    ['Bristol', 'Brighton', 14],
    ['Bristol', 'London', 13],
    ['Brighton', 'Brighton', 5],
    ['Brighton', 'London', 9],
    ['London', 'London', 10]], columns=['i', 'j', 'cost']).set_index(
        ['i', 'j'])

max_num_depts = 3

benefit = {}
for city in CITIES:
    for dept in DEPTS:
        try:
            benefit[dept, city] = benefit_data.ix[city, dept]
        except:
            benefit[dept, city] = 0

comm = {}
for row in comm_data.iterrows():
    (i, j) = row[0]
    comm[i, j] = row[1]['comm']
    comm[j, i] = comm[i, j]

cost = {}
for row in cost_data.iterrows():
    (i, j) = row[0]
    cost[i, j] = row[1]['cost']
    cost[j, i] = cost[i, j]

assign = m.add_variables(DEPTS, CITIES, vartype=so.BIN, name='assign')
IJKL = [(i, j, k, l)
        for i in DEPTS for j in CITIES for k in DEPTS for l in CITIES
        if i < k]
product = m.add_variables(IJKL, vartype=so.BIN)

totalBenefit = so.quick_sum(benefit[i, j] * assign[i, j]
```

(continues on next page)

(continued from previous page)

```
                                          for i in DEPTS for j in CITIES)

    totalCost = so.quick_sum(comm[i, k] * cost[j, l] * product[i, j, k, l]
                             for (i, j, k, l) in IJKL)

    m.set_objective(totalBenefit-totalCost, name='netBenefit', sense=so.MAX)

    m.add_constraints((so.quick_sum(assign[dept, city] for city in CITIES)
                      == 1 for dept in DEPTS), name='assign_dept')

    m.add_constraints((so.quick_sum(assign[dept, city] for dept in DEPTS)
                      <= max_num_depts for city in CITIES), name='cardinality')

    product_def1 = m.add_constraints((assign[i, j] + assign[k, l] - 1
                                     <= product[i, j, k, l]
                                     for (i, j, k, l) in IJKL),
                                     name='product_def1')

    product_def2 = m.add_constraints((product[i, j, k, l] <= assign[i, j]
                                      for (i, j, k, l) in IJKL),
                                     name='product_def2')

    product_def3 = m.add_constraints((product[i, j, k, l] <= assign[k, l]
                                      for (i, j, k, l) in IJKL),
                                     name='product_def3')

    m.solve()
    print(m.get_problem_summary())

    m.drop_constraints(product_def1)
    m.drop_constraints(product_def2)
    m.drop_constraints(product_def3)

    m.add_constraints((
        so.quick_sum(product[i, j, k, l]
                     for j in CITIES if (i, j, k, l) in IJKL) == assign[k, l]
        for i in DEPTS for k in DEPTS for l in CITIES if i < k),
        name='product_def4')

    m.add_constraints((
        so.quick_sum(product[i, j, k, l]
                     for l in CITIES if (i, j, k, l) in IJKL) == assign[i, j]
        for k in DEPTS for i in DEPTS for j in CITIES if i < k),
        name='product_def4')

    m.solve()
    print(m.get_problem_summary())
    totalBenefit.set_name('totalBenefit')
    totalCost.set_name('totalCost')
    print(so.get_solution_table(totalBenefit, totalCost))
    print(so.get_solution_table(assign).unstack(level=-1))

    return m.get_objective_value()
```

Output

```
>>> from examples.food_manufacture_1 import test
>>> sas_session = saspy.SASsession(cfgname='winlocal')
>>> test(sas_session)

SAS Connection established. Subprocess id is 14868

NOTE: Initialized model decentralization.
NOTE: Converting model decentralization to DataFrame.
NOTE: Writing HTML5(SASPY_INTERNAL) Body file: _TOMODS1
NOTE: The problem decentralization has 105 variables (105 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 278 constraints (183 LE, 5 EQ, 90 GE, 0 range).
NOTE: The problem has 660 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 120 constraints.
NOTE: The MILP presolver removed 120 constraint coefficients.
NOTE: The MILP presolver added 120 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 105 variables, 158 constraints, and 540 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      3     14.9000000     67.5000000   77.93%       0
             0        1      3     14.9000000     53.5000000   72.15%       0
             0        1      3     14.9000000     47.7000000   68.76%       0
             0        1      3     14.9000000     45.3000000   67.11%       0
             0        1      3     14.9000000     38.5000000   61.30%       0
             0        1      3     14.9000000     34.4666667   56.77%       0
             0        0      3     14.9000000     14.9000000    0.00%       0
NOTE: The MILP solver added 44 cuts with 254 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The data set WORK.PROB_SUMMARY has 21 observations and 3 variables.
NOTE: The data set WORK.SOL_SUMMARY has 17 observations and 3 variables.
NOTE: There were 825 observations read from the data set WORK.MPS.
NOTE: The data set WORK.PRIMAL_OUT has 105 observations and 8 variables.
NOTE: The data set WORK.DUAL_OUT has 278 observations and 8 variables.
NOTE: PROCEDURE OPTMILP used (Total process time):
      real time           0.09 seconds
      cpu time            0.06 seconds

                                    Value
Label
Problem Name                 decentralization
Objective Sense                  Maximization
Objective Function                 netBenefit
RHS                                       RHS

Number of Variables                       105
Bounded Above                               0
Bounded Below                               0
Bounded Above and Below                   105
```

(continues on next page)

(continued from previous page)

```
Free                                0
Fixed                               0
Binary                            105
Integer                             0

Number of Constraints             278
LE (<=)                           183
EQ (=)                              5
GE (>=)                            90
Range                               0

Constraint Coefficients           660
NOTE: Converting model decentralization to DataFrame.
NOTE: Writing HTML5(SASPY_INTERNAL) Body file: _TOMODS1
NOTE: The problem decentralization has 105 variables (105 binary, 0 integer, 0 free,␣
↪0 fixed).
NOTE: The problem has 68 constraints (3 LE, 65 EQ, 0 GE, 0 range).
NOTE: The problem has 270 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 105 variables, 68 constraints, and 270 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      3    -12.3000000    135.0000000  109.11%       0
             0        1      3    -12.3000000     30.0000000  141.00%       0
             0        1      3    -12.3000000     28.5000000  143.16%       0
             0        1      4     14.9000000     14.9000000    0.00%       0
NOTE: The MILP solver added 1 cuts with 2 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The data set WORK.PROB_SUMMARY has 21 observations and 3 variables.
NOTE: The data set WORK.SOL_SUMMARY has 17 observations and 3 variables.
NOTE: There were 384 observations read from the data set WORK.MPS.
NOTE: The data set WORK.PRIMAL_OUT has 105 observations and 8 variables.
NOTE: The data set WORK.DUAL_OUT has 68 observations and 8 variables.
NOTE: PROCEDURE OPTMILP used (Total process time):
      real time           0.08 seconds
      cpu time            0.06 seconds

                                  Value
Label
Problem Name             decentralization
Objective Sense               Maximization
Objective Function              netBenefit
RHS                                    RHS

Number of Variables               105
Bounded Above                       0
Bounded Below                       0
Bounded Above and Below           105
Free                                0
```

(continues on next page)

(continued from previous page)

```
Fixed                                   0
Binary                                105
Integer                                 0

Number of Constraints                  68
LE (<=)                                 3
EQ (=)                                 65
GE (>=)                                 0
Range                                   0

Constraint Coefficients               270
   totalBenefit  totalCost
1
           80.0       65.1
    assign  assign assign
2 Brighton Bristol London
1
A      0.0     1.0    0.0
B      1.0     0.0    0.0
C      1.0     0.0    0.0
D      0.0     1.0    0.0
E      1.0     0.0    0.0
SAS Connection terminated. Subprocess id was 14868
```

PYTHON MODULE INDEX

S

INDEX

S

T

U

V