sasoptpy Documentation

Release 0.2.1

SAS Institute Inc.

Feb 26, 2019

CONTENTS

PDF Version

Date: Feb 26, 2019 **Version**: 0.2.1

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

sasoptpy can handle linear optimization, mixed integer linear optimization, and nonlinear optimization problems. Users can benefit from native Python structures like dictionaries, tuples, and lists to define an optimization problem. *sasoptpy* supports Pandas objects extensively.

Under the hood, *sasoptpy* uses the swat package to communicate with SAS Viya, and uses the saspy package to communicate with 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

OVERVIEW

1.1 What's New

This page outlines changes from each release.

1.1.1 v0.2.1 (February 26, 2019)

New Features

- Support for evaluating nonlinear expressions is added, see `Expression.get_value()` and `utils._evaluate()`
- Support for multiple objectives is added for LSO solver, see `Model.set_objective()` and *Multiobjective* example
- Support for spaces inside variable indices is added
- Experimental RESTful API is added

Changes

- Dictionaries inside components are replaced with ordered dictionaries to preserve deterministic behavior
- Math operators are added into the keys of linear coefficient dictionaries
- Some iterators are rewritten using *yield* keyword for performance
- *key_name* and *col_names* parameters are added into `read_table()`

Bug Fixes

- Fixed: Using a single variable as an objective is producing incorrect input
- Fixed: `Expression.get_value()` fails to evaluate expressions with operators
- Fixed: `Expression.add()` overrides operators in some instances
- Fixed: Expressions with same components but different operators get summed incorrectly
- Fixed: New version of Viya complains about `pandas.DataFrame` column types
- Syntax fixes for **PEP 8** compliance

Notes

- A Jupyter notebook example of the Diet Problem is added
- A new example is added to show usage of experiment RESTful API: *Knapsack*
- Unit tests are added for development repository
- CD/CI integration is added for the development repository on Gitlab
- Generated models can be checked using the hash values inside tests.responses

1.1.2 v0.2.0 (July 30, 2018)

New Features

- Support for the new *runOptmodel* CAS action is added
- Nonlinear optimization model building support is added for both SAS 9.4 and SAS Viya solvers
- Abstract model building support is added when using SAS Viya solvers
- New object types, `Set`, `SetIterator`, `Parameter`, `ParameterValue`, `ImplicitVar`, `ExpressionDict`, and `Statement` are added for abstract model building
- `Model.to_optmodel()` method is added for exporting model objects into PROC OPTMODEL codes as a string
- Wrapper functions `read_table()` and `read_data()` are added to read CASTable and DataFrame objects into the models
- Math function wrappers are added
- `_expr` and `_defn` methods are added to all object types for producing OPTMODEL expression and definitions
- Multiple solutions are now being returned when using *solveMilp* action and can be grabbed using `Model.get_solution()` method
- `Model.get_variable_value()` is added to get solution values of abstract variables

Changes

- Variable and constraint naming schemes are replaced with OPTMODEL equivalent versions
- Variables and constraints now preserve the order they are inserted to the problem
- `Model.to_frame()` method is updated to reflect changes to VG and CG orderings
- Two solve methods, `Model.solve_on_cas()` and `Model.solve_on_viya()` are merged into `Model.solve()`
- `Model.solve()` method checks the available CAS actions and uses *runOptmodel* whenever possible
- As part of the merging process, `lp` and `milp` arguments are replaced with `options` argument in `Model.solve()` and `Model.to_optmodel()`
- An optional argument `frame` is added to `Model.solve()` for forcing to use MPS mode and *solveLp-solveMilp* actions
- Minor changes are applied to `__str__` and `__repr__` methods
- Creation indices for objects are being kept using the return of the `register_name()` function

- Objective constant values are now being passed using new CAS action arguments when posssible
- A linearity check is added for models
- Test folder is added to the repository

Bug Fixes

- Nondeterministic behavior when generating MPS files is fixed.

Notes

- Abstract and nonlinear models can be solved on Viya only if *runOptmodel* action is available on the CAS server.
- Three new examples are added which demonstrate abstract model building.
- Some minor changes are applied to the existing examples.

1.1.3 v0.1.2 (April 24, 2018)

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

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

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

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.

1.1.4 v0.1.1 (February 26, 2018)

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

Changes

- methods.py is renamed to utils.py

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

1.1.5 v0.1.0 (December 22, 2017)

- Initial release

1.2 Installation

1.2.1 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

1.2.2 Getting sasoptpy

sasoptpy can be installed using *pip* or *conda*:

```
pip install sasoptpy

conda install -c sas-institute sasoptpy
```

Any dependencies should be installed automatically.

GitHub repository

You can also get stable and development versions of *sasoptpy* from the GitHub repository. To get the latest version, 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
```

1.3 License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.
```

(continues on next page)

(continued from previous page)

```
      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
      copyright notice that is included in or attached to the work
      (an example is provided in the Appendix below).

      "Derivative Works" shall mean any work, whether in Source or Object
      form, that is based on (or derived from) the Work and for which the
      editorial revisions, annotations, elaborations, or other modifications
      represent, as a whole, an original work of authorship. For the purposes
      of this License, Derivative Works shall not include works that remain
      separable from, or merely link (or bind by name) to the interfaces of,
      the Work and Derivative Works thereof.

      "Contribution" shall mean any work of authorship, including
      the original version of the Work and any modifications or additions
      to that Work or Derivative Works thereof, that is intentionally
      submitted to Licensor for inclusion in the Work by the copyright owner
      or by an individual or Legal Entity authorized to submit on behalf of
      the copyright owner. For the purposes of this definition, "submitted"
      means any form of electronic, verbal, or written communication sent
      to the Licensor or its representatives, including but not limited to
      communication on electronic mailing lists, source code control systems,
      and issue tracking systems that are managed by, or on behalf of, the
      Licensor for the purpose of discussing and improving the Work, but
      excluding communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a Contribution."

      "Contributor" shall mean Licensor and any individual or Legal Entity
      on behalf of whom a Contribution has been received by Licensor and
      subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
```

(continues on next page)

(continued from previous page)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
```

(continues on next page)

(continued from previous page)

CHAPTER

TWO

GETTING STARTED

Solving an optimization problem via *sasoptpy* starts with having a running CAS Server or having a SAS 9.4 installation. It is possible to model a problem without a server but solving a problem requires access to SAS/OR solvers.

2.1 Creating a session

2.1.1 Creating a SAS Viya 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 use cases. See swat Documentation for more details.

2.1.2 Creating a SAS 9.4 session

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

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

2.2 Initializing a model

After having an active CAS/SAS session, 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.

2.3 Processing input data

The easisest 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']
```

2.4 Adding variables

You can add a single variables or a set of variables to *`Model`* objects.

- *`Model.add_variable()`* method 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 add it to an existing model using

  ```
  >>> m.include(production_cap)
  ```

- *`Model.add_variables()`* method is used to add a set of variables.

  ```
  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')
  ```

2.5 Creating expressions

Expression objects keep 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.

2.6 Setting an objective function

Objective functions can be written in terms of 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 * production[Period1] + 10 *␣
↪production[Period2] + 10 * production[Period3] - 10 * production_cap, 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.

2.7 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[Period1] - production_cap <=  0,␣
↪production[Period2] - production_cap <=  0, production[Period3] - production_cap <=␣
↪ 0, ], name='capacity')
```

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

(continues on next page)

(continued from previous page)

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. Range constraints can be inserted using == and a list of 2 values representing lower and upper bounds.

2.8 Solving a problem

Defined problems can be simply sent to CAS server or SAS sesion by calling the `Model.solve()` method.

See the solution output to the problem.

```
In [19]: m.solve()
NOTE: Added action set 'optimization'.
NOTE: Converting model my_first_model to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 4 variables (0 free, 0 fixed).
NOTE: The problem has 0 binary and 4 integer variables.
NOTE: The problem has 6 linear constraints (6 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 9 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
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.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 4 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 6 rows and 4 columns.
Out[19]:
                     var   lb             ub  value  rc
0        production_cap -0.0  1.797693e+308   25.0 NaN
1  production[Period1]  5.0  1.797693e+308   25.0 NaN
2  production[Period2]  5.0  1.797693e+308   15.0 NaN
3  production[Period3] -0.0  1.797693e+308   25.0 NaN
```

At the end of the solve operation, the solver 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
```

(continues on next page)

(continued from previous page)

```
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
Presolve Time                0.00
Solution Time                0.02
```

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

2.9 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.

2.10 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

THREE

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.

3.1 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.

3.1.1 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.

3.1.2 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')
```

3.1.3 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')
```

3.1.4 Set

sasoptpy can work with data on the server and generate abstract expressions. For this purpose, you can use `Set` objects to represent PROC OPTMODEL sets.

```
In [19]: m2 = so.Model(name='m2')
NOTE: Initialized model m2.

In [20]: I = m2.add_set(name='I')

In [21]: u = m2.add_variables(I, name='u')

In [22]: print(I, u)
I Variable Group (u) [
  [I: u[I]]
]
```

See *Workflows* for more information on working with server-side models.

3.2 Data

sasoptpy can work with both client-side and server-side data. Here are some options to load data into the optimization models.

3.2.1 pandas DataFrame

`pandas.DataFrame` is the preferred object types when passing data into *sasoptpy* models.

```
In [23]: data = [
   ....:     ['clock', 8, 4, 3],
   ....:     ['mug', 10, 6, 5],
   ....:     ['headphone', 15, 7, 2],
   ....:     ['book', 20, 12, 10],
   ....:     ['pen', 1, 1, 15]
   ....:     ]
   ....:

In [24]: df = pd.DataFrame(data, columns=['item', 'value', 'weight', 'limit']).set_
↪index(['item'])

In [25]: get = so.VariableGroup(df.index, ub=df['limit'], name='get')

In [26]: print(get)
Variable Group (get) [
  [book: get[book]]
  [clock: get[clock]]
  [headphone: get[headphone]]
  [mug: get[mug]]
  [pen: get[pen]]
]
```

3.2.2 Dictionaries

Lists and dictionaries can be used in expressions and when creating variables.

```
In [27]: items = ['clock', 'mug', 'headphone', 'book', 'pen']
```

(continues on next page)

(continued from previous page)

```
In [28]: limits = {'clock': 3, 'mug': 5, 'headphone': 2, 'book': 10, 'pen': 15}

In [29]: get2 = so.VariableGroup(items, ub=limits, name='get2')

In [30]: print(get2)
Variable Group (get2) [
  [book: get2[book]]
  [clock: get2[clock]]
  [headphone: get2[headphone]]
  [mug: get2[mug]]
  [pen: get2[pen]]
]
```

3.2.3 CASTable

When a data is available on the server-side, a reference to the object can be passed. Note that, using `swat.cas.table.CASTable` and Abstract Data requires SAS Viya version 3.4.

```
In [31]: m2 = so.Model(name='m2', session=session)
NOTE: Initialized model m2.
```

```
In [32]: table = session.upload_frame(df)
NOTE: Cloud Analytic Services made the uploaded file available as table TMPK7PDR7LP␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPK7PDR7LP has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
```

```
In [33]: print(type(table), table)
<class 'swat.cas.table.CASTable'> CASTable('TMPK7PDR7LP', caslib='CASUSERHDFS(casuser)
↪')
```

```
In [34]: df = pd.DataFrame(data, columns=['item', 'value', 'weight', 'limit'])

In [35]: ITEMS, (value, weight, limit) = m2.read_table(df, key=['item'],
   ....:     key_type='str', columns=['value', 'weight', 'limit'])
   ....:

In [36]: get3 = m2.add_variables(ITEMS, name='get3')

In [37]: print(get3)
Variable Group (get3) [
  [book: get3[book]]
  [clock: get3[clock]]
  [headphone: get3[headphone]]
  [mug: get3[mug]]
  [pen: get3[pen]]
]
```

3.2.4 Abstract Data

If you would like to model your problem first and then load data, you can pass a string for the data sets that will be available later. See following:

```
In [38]: m3 = so.Model(name='m3', session=session)
NOTE: Initialized model m3.

In [39]: ITEMS, (limit) = m3.read_table('DF', key=['item'], key_type=['str'],
   ....:     columns=['limit'])
   ....:

In [40]: print(type(ITEMS), ITEMS)
<class 'sasoptpy.data.Set'> set_item
```

Notice that the key set is created as a reference. We can later solve the problem after having the data available with the same name, e.g. using the *upload_frame* function.

```
In [41]: session.upload_frame(df, casout='DF')
NOTE: Cloud Analytic Services made the uploaded file available as table DF in caslib␣
↪CASUSERHDFS(casuser).
NOTE: The table DF has been created in caslib CASUSERHDFS(casuser) from binary data␣
↪uploaded to Cloud Analytic Services.
Out[41]: CASTable('DF', caslib='CASUSERHDFS(casuser)')
```

3.3 Operations

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

```
In [42]: sd = [3, 5, 6]

In [43]: z = m.add_variables(3, name='z')
```

```
In [44]: print(z)
Variable Group (z) [
  [0: z[0]]
  [1: z[1]]
  [2: z[2]]
]
```

```
In [45]: print(repr(z))
sasoptpy.VariableGroup([0, 1, 2], name='z')
```

```
In [46]: e1 = z.mult(sd)

In [47]: print(e1)
3 * z[0] + 5 * z[1] + 6 * z[2]
```

```
In [48]: ps = pd.Series(sd)

In [49]: e2 = z.mult(ps)

In [50]: print(e2)
3 * z[0] + 5 * z[1] + 6 * z[2]
```

CHAPTER

FOUR

SESSIONS AND MODELS

4.1 Sessions

4.1.1 CAS Sessions

A swat.cas.connection.CAS session is needed to solve optimization problems with *sasoptpy* using SAS Viya OR solvers. 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))
```

4.1.2 SAS Sessions

A saspy.SASsession session is needed to solve optimization problems with *sasoptpy* using SAS/OR solvers on SAS 9.4 clients.

A sample SAS session can be created using the following commands.

```
>>> import sasoptpy as so
>>> import saspy
>>> sas_session = saspy.SASsession(cfgname='winlocal')
>>> m = so.Model(name='demo', session=sas_session)
>>> print(repr(m))
sasoptpy.Model(name='demo', session=saspy.SASsession(cfgname='winlocal'))
```

4.2 Models

4.2.1 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.
```

4.2.2 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 [0]
  Variables (1): [
    x
  ]
  Constraints (0): [
  ]
]

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

In [6]: print(m)
Model: [
  Name: model1
  Objective: MIN [0]
  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 [0]
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
    x + 2 * y <=  10
  ]
]
```

4.2.3 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 [0]
  Variables (2): [
    x
    y
  ]
  Constraints (0): [
  ]
]

In [12]: new_model.include(c1)

In [13]: print(new_model)
Model: [
  Name: new_model
  Objective: MIN [0]
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
    x + 2 * 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 [0]
  Variables (3): [
    x
    y
    z
  ]
  Constraints (1): [
    x + 2 * 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.

4.2.4 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 * y <=  10, name='c1'), sasoptpy.Constraint(2 * 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 * x - y >=  1
```

4.2.5 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 [0]
  Variables (1): [
    x
  ]
  Constraints (2): [
    x + 2 * y <=  10
    2 * x - y >=  1
  ]
]
```

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

In [24]: print(m)
Model: [
  Name: model1
  Objective: MIN [0]
  Variables (2): [
    x
    y
  ]
  Constraints (2): [
    x + 2 * y <=  10
    2 * 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 [0]
  Variables (2): [
    x
    y
  ]
  Constraints (0): [
  ]
]
```

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

In [29]: print(m)
Model: [
  Name: model1
  Objective: MIN [0]
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
    x + 2 * y <=  10
  ]
]
```

4.2.6 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 [0]
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
    x + 2 * y <=  10
  ]
]
```

Note that all variables and constraints are included by reference.

4.2.7 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'.
...
NOTE: Optimal.
NOTE: Objective = 124.343.
NOTE: The Dual Simplex solve time is 0.01 seconds.
```

4.2.8 Solve options

Solver Options

Both PROC OPTMODEL solve options and `solveLp`, `solveMilp` action options can be passed using `options` argument of the `Model.solve()` method.

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

The only special option for the `Model.solve()` method is `with`. If not passed, PROC OPTMODEL chooses a solver that depends on the problem type. Possible `with` options are listed in SAS/OR documentation: http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_optmodel_syntax11.htm&docsetVersion=14.3&locale=en#ormpug.optmodel.npxsolvestmt

See specific solver options at following links:

- See http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_lpsolver_syntax02.htm&docsetVersion=14.3&locale=en for a list of LP solver options.
- See http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_milpsolver_syntax02.htm&docsetVersion=14.3&locale=en for a list of MILP solver options.
- See http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_nlpsolver_syntax02.htm&docsetVersion=14.3&locale=en for a list of NLP solver options.
- See http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_qpsolver_syntax02.htm&docsetVersion=14.3&locale=en for a list of QP solver options.
- See http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_clpsolver_syntax01.htm&docsetVersion=14.3&locale=en for a list of CLP solver options.

The `options` argument can also pass `solveLp` and `solveMilp` action options when `frame=True` is used when calling the `Model.solve()` method.

- 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 the `options` argument, there are 7 arguments that can be passed into *`Model.solve()`* method:

- name: Name of the uploaded problem information
- 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
- submit: Option for calling the CAS / SAS action
- frame: Option for using frame (MPS) method (if False, it uses OPTMODEL)
- verbose: Option for printing the generated OPTMODEL code before solve

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

4.2.9 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 FIVE

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.

5.1 Expressions

`Expression` objects represent linear and nonlinear mathematical expressions in *sasoptpy*.

5.1.1 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 = 5 * sales - 3 * material, name='profit')
```

5.1.2 Nonlinear expressions

`Expression` objects are linear by default. It is possible to create nonlinear expressions, but there are some limitations.

```
In [3]: nonexp = sales ** 2 + (1 / material) ** 3

In [4]: print(nonexp)
(sales) ** (2) + ((1) / (material)) ** (3)
```

Currently, it is not possible to get or print values of nonlinear expressions. Moreover, if your model includes a nonlinear expression, you need to be using SAS Viya >= 3.4 or any SAS version for solving your problem.

For using mathematical operations, you need to import *sasoptpy.math* functions.

5.1.3 Mathematical expressions

sasoptpy provides mathematical functions for generating mathematical expressions to be used in optimization models.

You need to import *sasoptpy.math* to your code to start using these functions. A list of available mathematical functions are listed at *Math Functions*.

```
In [5]: import sasoptpy.math as sm

In [6]: newexp = sm.max(sales, 10) ** 2

In [7]: print(newexp._expr())
(max(sales , 10)) ^ (2)
```

```
In [8]: import sasoptpy.math as sm

In [9]: angle = so.Variable(name='angle')

In [10]: newexp = sm.sin(angle) ** 2 + sm.cos(angle) ** 2

In [11]: print(newexp._expr())
(sin(angle)) ^ (2) + (cos(angle)) ^ (2)
```

5.1.4 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 [12]: tax = 0.5

In [13]: profit_after_tax = profit - tax
```

```
In [14]: print(repr(profit_after_tax))
sasoptpy.Expression(exp = 5 * sales - 3 * material - 0.5, name=None)
```

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

```
In [15]: profit_after_tax = profit.add(tax, sign=-1)
```

```
In [16]: print(profit_after_tax)
5 * sales - 3 * material - 0.5
```

```
In [17]: print(repr(profit_after_tax))
sasoptpy.Expression(exp = 5 * sales - 3 * material - 0.5, 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 [18]: investment = profit.mult(0.2)

In [19]: print(investment)
sales - 0.6 * material
```

Summation

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

```
In [20]: import time

In [21]: x = m.add_variables(1000, name='x')
```

```
In [22]: t0 = time.time()

In [23]: e = so.quick_sum(2 * x[i] for i in range(1000))

In [24]: print(time.time()-t0)
0.022248506546020508
```

```
In [25]: t0 = time.time()

In [26]: f = sum(2 * x[i] for i in range(1000))

In [27]: print(time.time()-t0)
0.5896790027618408
```

5.1.5 Renaming an expression

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

```
In [28]: e = so.Expression(x[5] + 2 * x[6], name='e1')

In [29]: print(repr(e))
sasoptpy.Expression(exp = x[5] + 2 * x[6], name='e1')
```

```
In [30]: e.set_name('e2');

In [31]: print(repr(e))
sasoptpy.Expression(exp = x[5] + 2 * x[6], name='e2')
```

5.1.6 Copying an expression

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

```
In [32]: copy_profit = profit.copy(name='copy_profit')

In [33]: print(repr(copy_profit))
sasoptpy.Expression(exp = 5 * sales - 3 * material, name='copy_profit')
```

5.1.7 Temporary expressions

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

```
In [34]: new_profit = so.Expression(10 * sales - 2 * material, temp=True)

In [35]: print(repr(new_profit))
sasoptpy.Expression(exp = 10 * sales - 2 * material, name=None)
```

The expression can be modified inside a function:

```
In [36]: new_profit + 5
Out[36]: sasoptpy.Expression(exp = 10 * sales - 2 * material + 5, name=None)
```

```
In [37]: print(repr(new_profit))
sasoptpy.Expression(exp = 10 * sales - 2 * material + 5, 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 [38]: new_profit = so.Expression(10 * sales - 2 * material, temp=True)

In [39]: new_profit.set_permanent()
Out[39]: 'expr_14'

In [40]: tmp = new_profit + 5

In [41]: print(repr(new_profit))
sasoptpy.Expression(exp = 10 * sales - 2 * material, name='expr_14')
```

5.2 Objective Functions

5.2.1 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
```

5.2.2 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
```

5.3 Variables

5.3.1 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')
```

5.3.2 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.

5.3.3 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')
```

5.3.4 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')
```

5.3.5 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',]]
]
sasoptpy.VariableGroup(['Period1', 'Period2', 'Period3'],
name='production')
```

5.4 Constraints

5.4.1 Creating constraints

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

Creating a constraint outside a model

```
>>> c1 = so.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')
```

5.4.2 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')
```

5.4.3 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 constraints 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 constraints 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]
]
```

5.4.4 Range constraints

A range for an expression can be given using a list of two value (lower and upper bound) with an == sign:

```
>>> x = m.add_variable(name='x')
>>> y = m.add_variable(name='y')
>>> c1 = m.add_constraint(x + 2*y == [2,9], name='c1')
>>> print(repr(c1))
sasoptpy.Constraint( x + 2.0 * y  ==  [2, 9], name='c1')
```

CHAPTER

SIX

WORKFLOWS

sasoptpy can work both with client-side data and server-side data. Some limitations to the functionalities may apply in terms of which workflow is being used. In this part, overall flow of the package is explained.

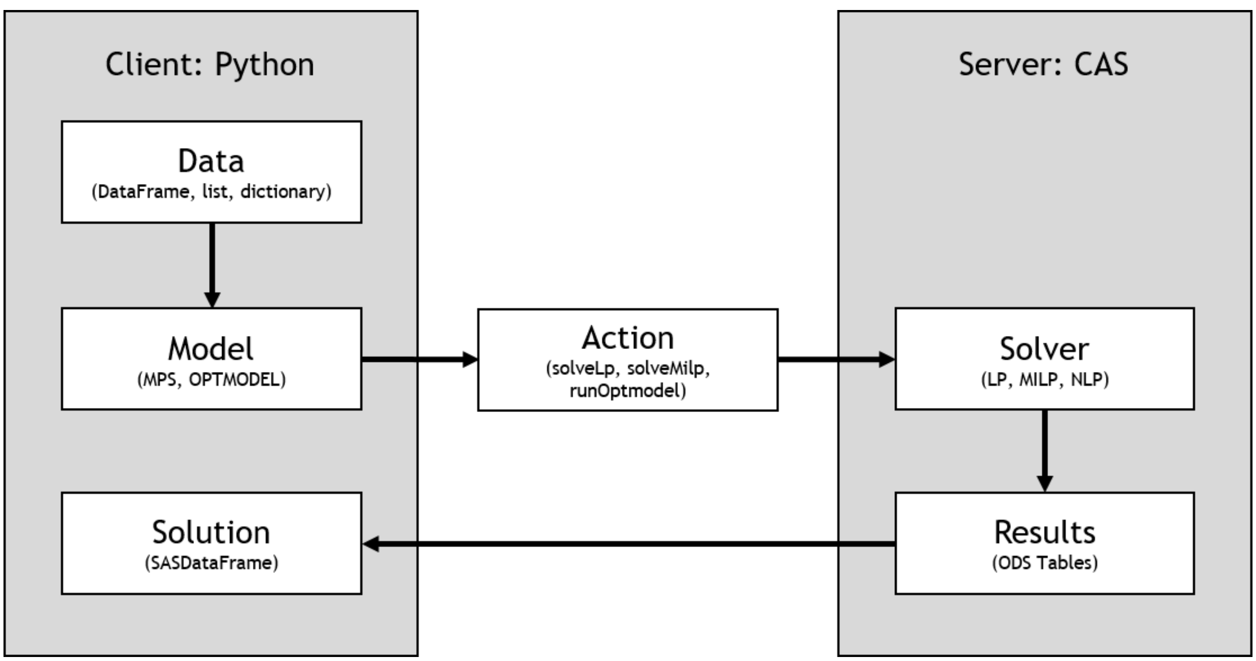
6.1 Client-side models

If the data is on the client-side (Python), then a concrete model is generated on the client-side and uploaded using one of the available CAS actions.

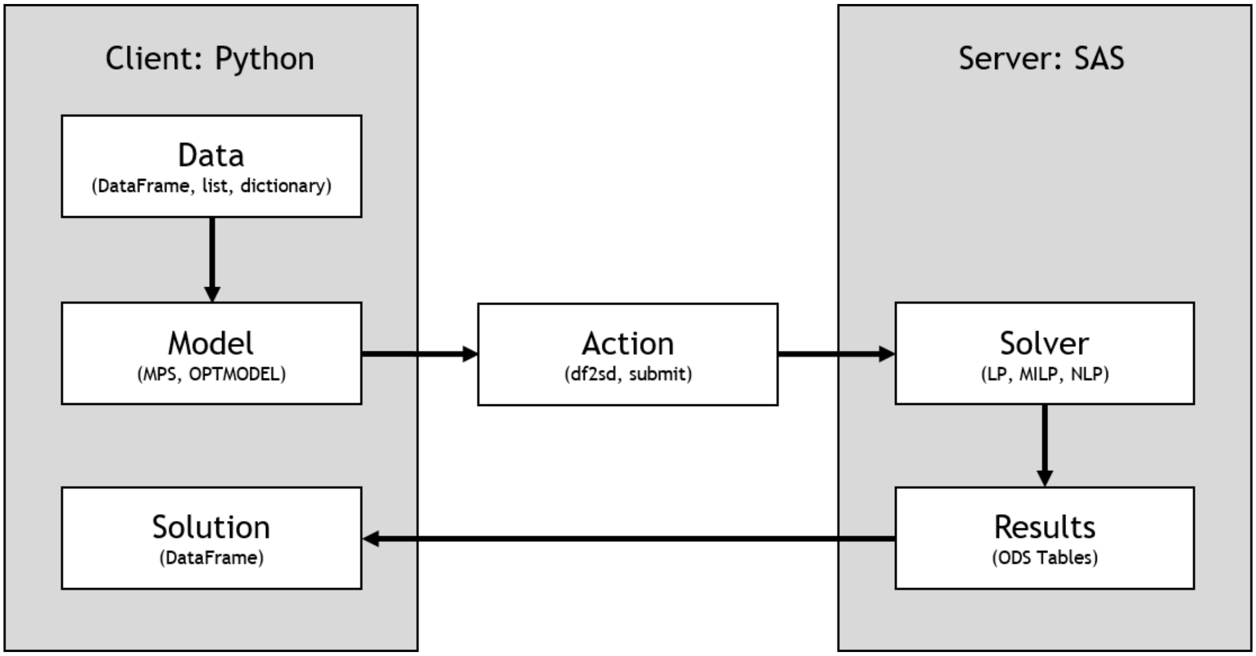
Using client-side models brings several advantages, such as accessing variables, expressions, and constraints directly. You may do more intensive operations like filter data, sort values, changing variable values, and print expressions more easily.

There are two main disadvantages of working with client-side models. First, if your model is relatively big size, then the generated MPS DataFrame or OPTMODEL codes may allocate a large memory on your machine. Second, the information that needs to be passed from client to server might be bigger than using a server-side model.

See the following representation of the client-side model workflow for CAS (Viya) servers:



See the following representation of the client-side model workflow for SAS clients:



Steps of modeling a simple Knapsack problem are shown in the following subsections.

6.1.1 Reading data

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

In [2]: import pandas as pd

In [3]: from swat import CAS

In [4]: session = CAS(hostname, port)

In [5]: m = so.Model(name='client_CAS', session=session)
NOTE: Initialized model client_CAS.

In [6]: data = [
   ...:     ['clock', 8, 4, 3],
   ...:     ['mug', 10, 6, 5],
   ...:     ['headphone', 15, 7, 2],
   ...:     ['book', 20, 12, 10],
   ...:     ['pen', 1, 1, 15]
   ...:     ]
   ...:

In [7]: df = pd.DataFrame(data, columns=['item', 'value', 'weight', 'limit'])

In [8]: ITEMS, (value, weight, limit) = m.read_table(
   ...:     df, key=['item'], columns=['value', 'weight', 'limit'])
   ...:

In [9]: total_weight = 55
```

```
In [10]: print(type(ITEMS), ITEMS)
<class 'list'> ['clock', 'mug', 'headphone', 'book', 'pen']
```

```
In [11]: print(type(total_weight), total_weight)
<class 'int'> 55
```

Here, instead of using *Model.read_table()* method, column values can be obtained one by one:

```
>>> df = df.set_index('item')
>>> ITEMS = df.index.tolist()
>>> value = df['value']
>>> weight = df['weight']
>>> limit = df['limit']
```

6.1.2 Model

```
# Variables
In [12]: get = m.add_variables(ITEMS, name='get', vartype=so.INT)

# Constraints
In [13]: m.add_constraints((get[i] <= limit[i] for i in ITEMS), name='limit_con');

In [14]: m.add_constraint(
   ....:     so.quick_sum(weight[i] * get[i] for i in ITEMS) <= total_weight,
   ....:     name='weight_con');
   ....:

# Objective
In [15]: total_value = so.quick_sum(value[i] * get[i] for i in ITEMS)

In [16]: m.set_objective(total_value, name='total_value', sense=so.MAX);

# Solve
In [17]: m.solve(verbose=True)
NOTE: Added action set 'optimization'.
NOTE: Converting model client_CAS to OPTMODEL.
var get {{'clock','mug','headphone','book','pen'}} integer >= 0;
con limit_con_clock : get['clock'] <= 3;
con limit_con_mug : get['mug'] <= 5;
con limit_con_headphone : get['headphone'] <= 2;
con limit_con_book : get['book'] <= 10;
con limit_con_pen : get['pen'] <= 15;

con weight_con : 4 * get['clock'] + 6 * get['mug'] + 7 * get['headphone'] + 12 * get[
↪'book'] + get['pen'] <= 55;
max total_value = 8 * get['clock'] + 10 * get['mug'] + 15 * get['headphone'] + 20 *␣
↪get['book'] + get['pen'];
solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 binary and 5 integer variables.
NOTE: The problem has 6 linear constraints (6 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 10 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
```

(continues on next page)

(continued from previous page)

```
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 5 constraints.
NOTE: The MILP presolver removed 5 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5 variables, 1 constraints, and 5 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 80 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      4     99.0000000    199.0000000   50.25%       0
                0        1      4     99.0000000    102.3333333    3.26%       0
                0        1      4     99.0000000    102.3333333    3.26%       0
NOTE: The MILP presolver is applied again.
                0        1      4     99.0000000    102.3333333    3.26%       3
NOTE: Optimal.
NOTE: Objective = 99.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 5 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 6 rows and 4 columns.
Out[17]:
              var   lb             ub  value  rc
0      get[clock] -0.0  1.797693e+308    2.0 NaN
1        get[mug] -0.0  1.797693e+308    5.0 NaN
2  get[headphone] -0.0  1.797693e+308    2.0 NaN
3       get[book] -0.0  1.797693e+308   -0.0 NaN
4        get[pen] -0.0  1.797693e+308    3.0 NaN
```

Using `verbose` option shows the generated OPTMODEL code. Here, we can see the coefficient values of the parameters inside the model.

6.1.3 Parsing results

After the solve, primal and dual solution tables are obtained. We can print the solution tables using the *Model.get_solution()* method.

It is also possible to print the optimal solution using the *get_solution_table()* function.

```
In [18]: print(m.get_solution())
              var   lb             ub  value  rc
0      get[clock] -0.0  1.797693e+308    2.0 NaN
1        get[mug] -0.0  1.797693e+308    5.0 NaN
2  get[headphone] -0.0  1.797693e+308    2.0 NaN
3       get[book] -0.0  1.797693e+308   -0.0 NaN
4        get[pen] -0.0  1.797693e+308    3.0 NaN
```

```
In [19]: print(so.get_solution_table(get, key=ITEMS))
           get
1
```

(continues on next page)

(continued from previous page)

```
clock       2.0
mug         5.0
headphone   2.0
book       -0.0
pen         3.0
```

```
In [20]: print('Total value:', total_value.get_value())
Total value: 99.0
```

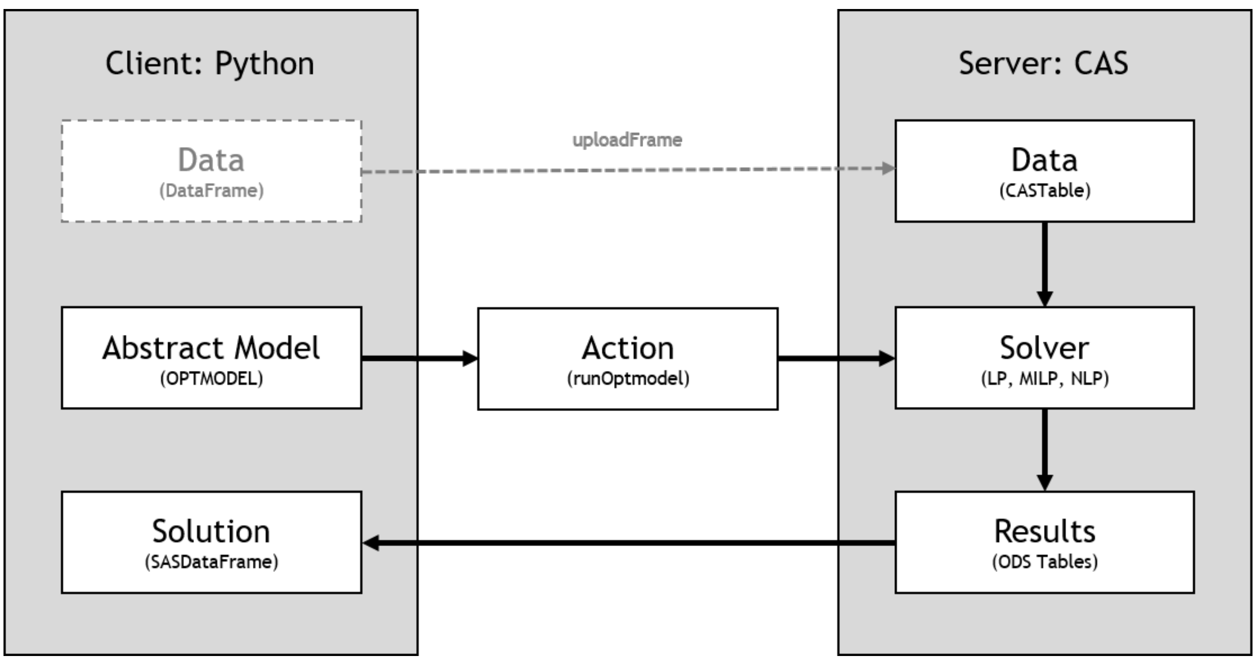
6.2 Server-side models

If the data is on the server-side (CAS or SAS), then an abstract model is generated on the client-side. This abstract model is later converted to PROC OPTMODEL code, which combines the data on the server.

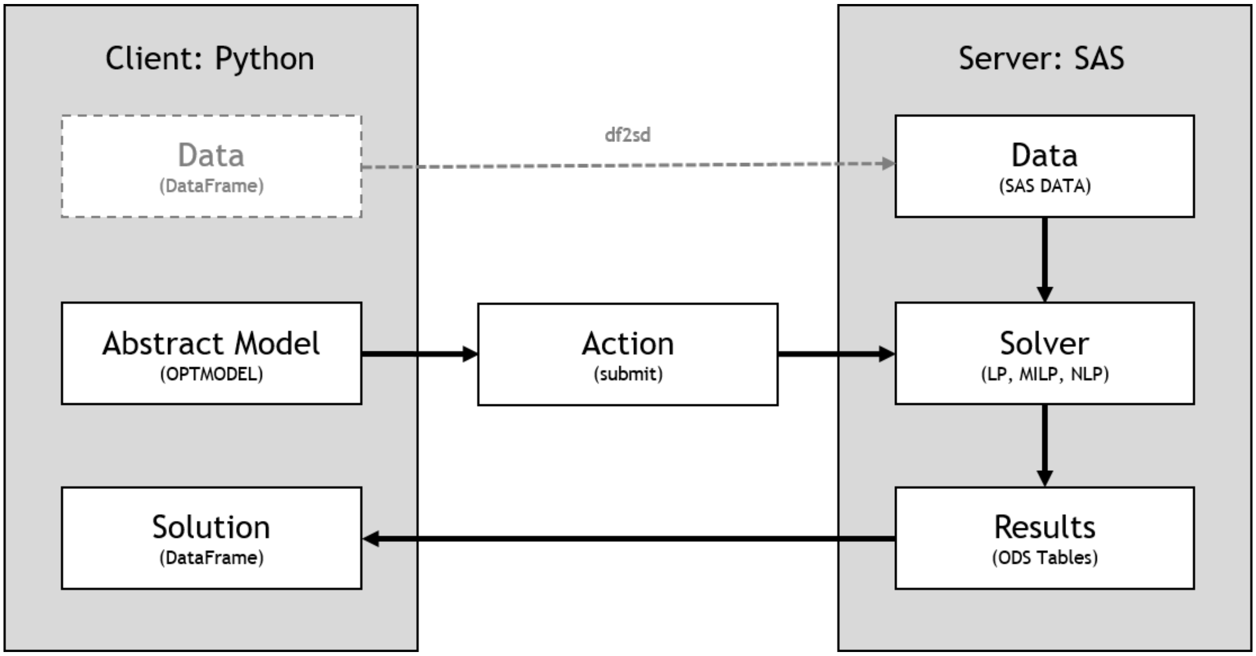
The main advantage of the server-side models is faster upload times compared to client-side. This is especially very noticable when using large chunks of variable and constraint groups.

The only disadvantage of using server-side models is that variables often need to be accessed directly from the resulting SASDataFrame objects. Since components of the models are abstract, accessing objects directly is often not possible.

See the following representation of the server-side model workflow for CAS (Viya) servers:



See the following representation of the server-side model workflow for SAS clients:



In the following subsections, the same example will be solved using server-side data.

6.2.1 Uploading data (Optional)

It is possible to upload client-side data to server-side when working with relatively big models.

sasoptpy supports using `swat.cas.table.CASTable` objects. The `swat.cas.connection.CAS.upload_frame()` method can be used to upload `pandas.DataFrame` objects to the CAS Server. Another way is to use *`read_table()`* function with `upload=True` option.

```
In [21]: session = CAS(hostname, port)

In [22]: m = so.Model(name='server_CAS', session=session)
NOTE: Initialized model server_CAS.

In [23]: data = [
   ....:     ['clock', 8, 4, 3],
   ....:     ['mug', 10, 6, 5],
   ....:     ['headphone', 15, 7, 2],
   ....:     ['book', 20, 12, 10],
   ....:     ['pen', 1, 1, 15]
   ....:     ]
   ....:

In [24]: df = pd.DataFrame(data, columns=['item', 'value', 'weight', 'limit'])

In [25]: ITEMS, (value, weight, limit) = m.read_table(
   ....:     df, key=['item'], key_type='str', columns=['value', 'weight', 'limit'],
   ....:     upload=True, casout='df')
   ....:
NOTE: Cloud Analytic Services made the uploaded file available as table DF in caslib␣
↪CASUSERHDFS(casuser).
NOTE: The table DF has been created in caslib CASUSERHDFS(casuser) from binary data␣
↪uploaded to Cloud Analytic Services.
```

(continues on next page)

(continued from previous page)

```
In [26]: total_weight = m.add_parameter(init = 55, name='total_weight')
```

```
In [27]: print(type(ITEMS), ITEMS)
<class 'sasoptpy.data.Set'> set_item
```

```
In [28]: print(type(total_weight), total_weight)
<class 'sasoptpy.data.ParameterValue'> total_weight
```

Since we use `upload=True` option, the data is uploaded to the server and we get a CASTable object. Similarly, `total_weight` is a parameter object here.

6.2.2 Model

```
# Variables
In [29]: get = m.add_variables(ITEMS, name='get', vartype=so.INT)

# Constraints
In [30]: m.add_constraints((get[i] <= limit[i] for i in ITEMS), name='limit_con');

In [31]: m.add_constraint(
   ....:     so.quick_sum(weight[i] * get[i] for i in ITEMS) <= total_weight,
   ....:     name='weight_con');
   ....:

# Objective
In [32]: total_value = so.quick_sum(value[i] * get[i] for i in ITEMS)

In [33]: m.set_objective(total_value, name='total_value', sense=so.MAX);

# Solve
In [34]: m.solve(verbose=True)
NOTE: Added action set 'optimization'.
NOTE: Converting model server_CAS to OPTMODEL.
set <str> set_item;
num value {set_item};
num weight {set_item};
num limit {set_item};
read data DF into set_item=[item] value weight limit;
num total_weight init 55;
var get {{set_item}} integer >= 0;
con limit_con {i_1 in set_item} : get[i_1] - limit[i_1] <= 0;

con weight_con : total_weight - sum {i_2 in set_item}(weight[i_2] * get[i_2]) >= 0;
max total_value = sum {i_3 in set_item}(value[i_3] * get[i_3]);
solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: There were 5 rows read from table 'DF' in caslib 'CASUSERHDFS(casuser)'.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 binary and 5 integer variables.
```

(continues on next page)

(continued from previous page)

```
NOTE: The problem has 6 linear constraints (5 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 10 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 5 constraints.
NOTE: The MILP presolver removed 5 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5 variables, 1 constraints, and 5 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      4     99.0000000    199.0000000   50.25%       0
                0        1      4     99.0000000    102.3333333    3.26%       0
                0        0      4     99.0000000     99.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 99.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 5 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 6 rows and 4 columns.
Out[34]:
              var   lb             ub  value  rc
0       get[book] -0.0  1.797693e+308    2.0 NaN
1      get[clock] -0.0  1.797693e+308    3.0 NaN
2  get[headphone] -0.0  1.797693e+308    2.0 NaN
3        get[mug] -0.0  1.797693e+308   -0.0 NaN
4        get[pen] -0.0  1.797693e+308    5.0 NaN
```

There is no difference in terms of how client-side and server-side models are written. However, the generated OPTMODEL code is more compact for server-side models.

6.2.3 Parsing results

```
# Print results
In [35]: print(m.get_solution())
              var   lb             ub  value  rc
0       get[book] -0.0  1.797693e+308    2.0 NaN
1      get[clock] -0.0  1.797693e+308    3.0 NaN
2  get[headphone] -0.0  1.797693e+308    2.0 NaN
3        get[mug] -0.0  1.797693e+308   -0.0 NaN
4        get[pen] -0.0  1.797693e+308    5.0 NaN
```

```
In [36]: print('Total value:', m.get_objective_value())
Total value: 99.0
```

Since there is no direct access to expressions and variables, the optimal solution is printed using the server response.

6.3 Limitations

- Nonlinear models can only be solved using runOptmodel action, hence requires SAS Viya version to be greater than or equal to 3.4.
- User defined decomposition blocks are only available in MPS mode, hence only works with client-side data.
- Mixed usage (client-side and server-side data) may not work in some cases. A quick fix would be transferring the data, in either direction.

CHAPTER

SEVEN

EXAMPLES

Examples are provided from SAS/OR documentation.

7.1 Viya Examples / Concrete

7.1.1 Food Manufacture 1

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex1_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex01.html

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, index=PERIODS).transpose()
    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
```

(continues on next page)

(continued from previous page)

```
    hardness_lb = 3
    hardness_ub = 6
    init_storage = 500

    # Problem initialization
    m = so.Model(name='food_manufacture_1', 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 = m.add_implicit_variable((use.sum('*', p) for p in PERIODS),
                                          name='manufacture')
    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] for p in PERIODS)
    rawcost = so.quick_sum(cost.at[o, p] * 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] 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] for p in PERIODS),
                      name='hardness_lb')

    # Solver call
    res = m.solve()

    # With other solve options
    m.solve(options={'with': 'lp', 'algorithm': 'PS'})
    m.solve(options={'with': 'lp', 'algorithm': 'IP'})
    m.solve(options={'with': '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: Added action set 'optimization'.
NOTE: Converting model food_manufacture_1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 linear constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 44 variables and 4 constraints.
NOTE: The LP presolver removed 48 constraint coefficients.
NOTE: The presolved problem has 51 variables, 50 constraints, and 162 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    4.755480E+05         0
          P 2         50    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 54 rows and 4 columns.
NOTE: Added action set 'optimization'.
NOTE: Converting model food_manufacture_1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 linear constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 44 variables and 4 constraints.
NOTE: The LP presolver removed 48 constraint coefficients.
NOTE: The presolved problem has 51 variables, 50 constraints, and 162 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          P 1          1    1.704920E+03         0
          P 2         34    9.938901E+03         0
```

(continues on next page)

(continued from previous page)

```
          D 2          53    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Primal Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 54 rows and 4 columns.
NOTE: Added action set 'optimization'.
NOTE: Converting model food_manufacture_1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 linear constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 44 variables and 4 constraints.
NOTE: The LP presolver removed 48 constraint coefficients.
NOTE: The LP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 51 variables, 50 constraints, and 162 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 80 threads.
                                                 Primal         Bound          Dual
         Iter   Complement Duality Gap       Infeas        Infeas        Infeas    Time
            0   1.1003E+04  1.3994E+01  2.0602E-02  1.1145E-02  1.2444E+00       0
            1   1.0498E+04  4.1015E+01  1.7385E-02  9.4051E-03  1.0928E+00       0
            2   7.2084E+03  7.4551E+00  5.6703E-03  3.0675E-03  6.8365E-01       0
            3   1.7518E+03  1.1221E+00  1.5798E-03  8.5465E-04  1.1852E-01       0
            4   4.1038E+02  2.5544E-01  5.6092E-04  3.0344E-04  1.1852E-03       0
            5   3.9774E+01  2.2775E-02  7.2994E-05  3.9488E-05  1.9281E-05       0
            6   9.9400E-01  5.6526E-04  7.9112E-07  4.2798E-07  7.7185E-07       0
            7   9.9572E-03  5.6615E-06  7.9420E-09  4.2964E-09  7.7239E-09       0
            8   0.0000E+00  1.8686E-08  1.6613E-07  1.1864E-10  6.2833E-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    1.144523E+03         0
          D C          13    1.071811E+01         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 0.02 seconds.
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 16 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 54 rows and 4 columns.
NOTE: Added action set 'optimization'.
NOTE: Converting model food_manufacture_1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 95 variables (0 free, 10 fixed).
NOTE: The problem has 54 linear constraints (18 LE, 30 EQ, 6 GE, 0 range).
NOTE: The problem has 210 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 44 variables and 4 constraints.
NOTE: The LP presolver removed 48 constraint coefficients.
NOTE: The presolved problem has 51 variables, 50 constraints, and 162 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 38 rows (76.00%), 51 columns (100.00%), and 2 components.
NOTE: The network extraction and setup time is 0.00 seconds.
         Component     Nodes      Arcs   Iterations     Time
                 1        18        22           18     0.00
                 2        20        29           24     0.00
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.082500E+05         0
          P 2          8    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 14 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 54 rows and 4 columns.
           buy          use       store
1    2
oil1 0       -            -  500.000000
oil1 1       0            0  500.000000
oil1 2       0  1.89478e-14  500.000000
oil1 3       0            0  500.000000
oil1 4       0            0  500.000000
oil1 5       0            0  500.000000
oil1 6       0            0  500.000000
oil2 0       -            -  500.000000
oil2 1       0 -2.84217e-14  500.000000
oil2 2     250            0  750.000000
oil2 3       0          250  500.000000
oil2 4       0          250  250.000000
oil2 5       0          250    0.000000
oil2 6     750          250  500.000000
oil3 0       -            -  500.000000
```

(continues on next page)

(continued from previous page)

```
oil3 1        0         250  250.000000
oil3 2        0         250    0.000000
oil3 3        0           0    0.000000
oil3 4        0           0    0.000000
oil3 5      500           0  500.000000
oil3 6        0           0  500.000000
veg1 0        -           -  500.000000
veg1 1        0     85.1852  414.814815
veg1 2        0     85.1852  329.629630
veg1 3        0     159.259  170.370370
veg1 4        0     11.1111  159.259259
veg1 5        0     159.259    0.000000
veg1 6  659.259     159.259  500.000000
veg2 0        -           -  500.000000
veg2 1        0     114.815  385.185185
veg2 2        0     114.815  270.370370
veg2 3        0     40.7407  229.629630
veg2 4        0     188.889   40.740741
veg2 5        0     40.7407    0.000000
veg2 6  540.741     40.7407  500.000000
Out[2]: 107842.592593
```

7.1.2 Food Manufacture 2

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex2_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex02.html

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, index=PERIODS).transpose()
    hardness_data = [8.8, 6.1, 2.0, 4.2, 5.0]
    hardness = {OILS[i]: hardness_data[i] for i in range(len(OILS))}
```

(continues on next page)

(continued from previous page)

```
    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 = m.add_implicit_variable((use.sum('*', p) for p in PERIODS),
                                          name='manufacture')
    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] for p in PERIODS)
    rawcost = so.quick_sum(cost.at[o, p] * 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] 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] 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)
```

(continues on next page)

(continued from previous page)

```
    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: Added action set 'optimization'.
NOTE: Converting model food_manufacture_2 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 125 variables (0 free, 10 fixed).
NOTE: The problem has 30 binary and 0 integer variables.
NOTE: The problem has 132 linear constraints (66 LE, 30 EQ, 36 GE, 0 range).
NOTE: The problem has 384 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
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.
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  29000.0000000         343250   91.55%       0
                0        1      3  29000.0000000         107333   72.98%       0
                0        1      3  29000.0000000         105799   72.59%       0
                0        1      3  29000.0000000         105650   72.55%       0
                0        1      3  29000.0000000         105650   72.55%       0
                0        1      3  29000.0000000         105650   72.55%       0
                0        1      3  29000.0000000         105650   72.55%       0
                0        1      3  29000.0000000         105650   72.55%       0
                0        1      4  44000.0000000         105650   58.35%       0
NOTE: The MILP solver added 22 cuts with 99 cut coefficients at the root.
                1        2      5  64446.4285714         105650   39.00%       0
                2        3      6  73921.4285714         105650   30.03%       0
```

(continues on next page)

(continued from previous page)

```
                 51        36       7          100279          104920     4.42%        1
                301         2       8          100279          100728     0.45%        2
                304         0       8          100279          100279     0.00%        2
NOTE: Optimal.
NOTE: Objective = 100278.7037.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 125 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 132 rows and 4␣
↪columns.
                  buy          use       store is_used
1    2
oil1 0            -            -  500.000000       -
oil1 1            0            0  500.000000       0
oil1 2            0   4.0172e-14  500.000000       0
oil1 3            0            0  500.000000      -0
oil1 4  1.13687e-13            0  500.000000       0
oil1 5            0  4.54747e-14  500.000000       0
oil1 6            0            0  500.000000       0
oil2 0            -            -  500.000000       -
oil2 1            0            0  500.000000      -0
oil2 2            0            0  500.000000       0
oil2 3            0           40  460.000000       1
oil2 4            0          230  230.000000       1
oil2 5            0          230    0.000000       1
oil2 6          730          230  500.000000       1
oil3 0            -            -  500.000000       -
oil3 1            0          250  250.000000       1
oil3 2            0          250    0.000000       1
oil3 3          770          210  560.000000       1
oil3 4            0           20  540.000000       1
oil3 5            0           20  520.000000       1
oil3 6            0           20  500.000000       1
veg1 0            -            -  500.000000       -
veg1 1            0      85.1852  414.814815       1
veg1 2            0      85.1852  329.629630       1
veg1 3            0            0  329.629630       0
veg1 4            0          155  174.629630       1
veg1 5            0          155   19.629630       1
veg1 6       480.37            0  500.000000       0
veg2 0            -            -  500.000000       -
veg2 1            0      114.815  385.185185       1
veg2 2            0      114.815  270.370370       1
veg2 3 -1.42109e-14          200   70.370370       1
veg2 4            0            0   70.370370       0
veg2 5            0            0   70.370370       0
veg2 6       629.63          200  500.000000       1
Out[2]: 100278.703704
```

7.1.3 Factory Planning 1

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex3_toc.htm&docsetVersion=14.3&locale=en

https://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex03.html

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'], 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, index=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 = [
        ['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],
```

(continues on next page)

(continued from previous page)

```
        ['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),
                      name='flow_balance')

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

    return m.get_objective_value()
```

Output

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

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

(continues on next page)

(continued from previous page)

```
NOTE: Initialized model factory_planning_1.
NOTE: Added action set 'optimization'.
NOTE: Converting model factory_planning_1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 126 variables (0 free, 6 fixed).
NOTE: The problem has 72 linear constraints (30 LE, 42 EQ, 0 GE, 0 range).
NOTE: The problem has 281 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 60 variables and 46 constraints.
NOTE: The LP presolver removed 178 constraint coefficients.
NOTE: The presolved problem has 66 variables, 26 constraints, and 103 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.478510E+04         0
          P 2         21    9.371518E+04         0
NOTE: Optimal.
NOTE: Objective = 93715.178571.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 126 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 72 rows and 4 columns.
                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
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
```

(continues on next page)

(continued from previous page)

```
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
Out[2]: 93715.178571
```

7.1.4 Factory Planning 2

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex4_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex04.html

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'], 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, index=range(1, 7))
```

(continues on next page)

(continued from previous page)

```
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

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

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')
```

(continues on next page)

(continued from previous page)

```
    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: Added action set 'optimization'.
NOTE: Converting model factory_planning_2 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 156 variables (0 free, 13 fixed).
NOTE: The problem has 0 binary and 30 integer variables.
NOTE: The problem has 77 linear constraints (30 LE, 47 EQ, 0 GE, 0 range).
NOTE: The problem has 341 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
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  92755.0000000         116455   20.35%       0
                0        1      2  92755.0000000         116455   20.35%       0
                0        1      2  92755.0000000         116141   20.14%       0
                0        1      2  92755.0000000         115660   19.80%       0
                0        1      2  92755.0000000         114136   18.73%       0
                0        1      2  92755.0000000         113334   18.16%       0
                0        1      2  92755.0000000         112487   17.54%       0
```

(continues on next page)

(continued from previous page)

```
                 0         1       2   92755.0000000         111392   16.73%       0
                 0         1       2   92755.0000000         111128   16.53%       0
                 0         1       2   92755.0000000         110036   15.71%       0
                 0         1       2   92755.0000000         109206   15.06%       0
                 0         1       2   92755.0000000         109049   14.94%       0
                 0         1       2   92755.0000000         108975   14.88%       0
                 0         1       2   92755.0000000         108855   14.79%       0
                 0         1       2   92755.0000000         108855   14.79%       0
                 0         1       3          108855         108855    0.00%       0
NOTE: The MILP solver added 43 cuts with 168 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 108855.00023.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 156 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 77 rows and 4 columns.
                  make          sell         store
1      2
prod1 1  5.000001e+02   500.000000  1.135733e-04
prod1 2  6.000001e+02   599.999865  3.390655e-04
prod1 3  3.999995e+02   300.000000  9.999983e+01
prod1 4  3.385140e-04   100.000169  0.000000e+00
prod1 5  0.000000e+00     0.000000  0.000000e+00
prod1 6  5.500000e+02   500.000000  5.000000e+01
prod2 1  1.000000e+03  1000.000000  0.000000e+00
prod2 2  4.999999e+02   499.999775  1.028513e-04
prod2 3  6.999992e+02   599.999435  9.999983e+01
prod2 4  5.077710e-04   100.000339  0.000000e+00
prod2 5  1.000000e+02   100.000000 -1.667049e-13
prod2 6  5.500000e+02   500.000000  5.000000e+01
prod3 1  3.000000e+02   300.000000  0.000000e+00
prod3 2  2.000000e+02   200.000000  0.000000e+00
prod3 3  9.999904e+01     0.000000  9.999904e+01
prod3 4  1.301130e-03   100.000339  0.000000e+00
prod3 5  5.000000e+02   500.000000  0.000000e+00
prod3 6  1.500000e+02   100.000000  5.000000e+01
prod4 1  3.000000e+02   300.000000  0.000000e+00
prod4 2  0.000000e+00     0.000000  0.000000e+00
prod4 3  9.999982e+01     0.000000  9.999982e+01
prod4 4  8.562850e-04   100.000677  0.000000e+00
prod4 5  1.000000e+02   100.000000  0.000000e+00
prod4 6  3.500000e+02   300.000000  5.000000e+01
prod5 1  8.000000e+02   800.000000  0.000000e+00
prod5 2  4.000002e+02   400.000000  2.281596e-04
prod5 3  5.999993e+02   499.999548  1.000000e+02
prod5 4  1.692570e-04   100.000169 -2.664116e-13
prod5 5  1.000000e+03  1000.000000 -9.849305e-14
prod5 6  1.150000e+03  1100.000000  5.000000e+01
prod6 1  2.000000e+02   200.000000  0.000000e+00
prod6 2  3.000000e+02   300.000000  0.000000e+00
prod6 3  4.000000e+02   400.000000  0.000000e+00
prod6 4  0.000000e+00     0.000000  0.000000e+00
prod6 5  3.000000e+02   300.000000  0.000000e+00
prod6 6  5.500000e+02   500.000000  5.000000e+01
```

(continues on next page)

(continued from previous page)

```
prod7 1  1.000000e+02   100.000000 -8.526513e-14
prod7 2  1.500001e+02   150.000000  1.130218e-04
prod7 3  1.999999e+02   100.000000  1.000000e+02
prod7 4  0.000000e+00   100.000000  0.000000e+00
prod7 5 -2.366403e-12     0.000000 -2.366403e-12
prod7 6  1.100000e+02    60.000000  5.000000e+01
        numMachinesDown numMachinesDown numMachinesDown numMachinesDown␣
↪numMachinesDown numMachinesDown
2                     1               2               3                4                ␣
↪5              6
1                                                                                       ␣
↪
borer              -0.0    5.623517e-07        0.000001        0.999998   -6.996782e-
↪16   -1.175077e-15
grinder            -0.0   -0.000000e+00       -0.000000        2.000000   -0.
↪000000e+00   -0.000000e+00
hdrill              1.0    2.000000e+00       -0.000000       -0.000000   -0.
↪000000e+00   -0.000000e+00
planer              0.0    0.000000e+00        0.000001        0.999999   -4.657779e-
↪16    0.000000e+00
vdrill              0.0   -0.000000e+00       -0.000000        1.999998    1.792570e-
↪06   -1.139926e-15
Solution Summary

                                             Value
Label
Solver                                        MILP
Algorithm                           Branch and Cut
Objective Function                    total_profit
Solution Status       Optimal within Relative Gap
Objective Value                    108855.00023

Relative Gap                          1.4193953E-8
Absolute Gap                          0.0015450828
Primal Infeasibility                  1.065814E-13
Bound Infeasibility                   2.366403E-12
Integer Infeasibility                 1.7925699E-6

Best Bound                         108855.00178
Nodes                                            1
Solutions Found                                  3
Iterations                                     362
Presolve Time                                 0.00
Solution Time                                 1.21
Problem Summary

                                Value
Label
Objective Sense           Maximization
Objective Function        total_profit
Objective Type                  Linear

Number of Variables                156
Bounded Above                        0
Bounded Below                       72
Bounded Below and Above             71
Free                                 0
```

(continues on next page)

(continued from previous page)

```
Fixed                              13
Binary                              0
Integer                            30

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

Constraint Coefficients           341
Out[2]: 108855.000235
```

7.1.5 Manpower Planning

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex5_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex05.html

Model

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


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

(continues on next page)

(continued from previous page)

```
        ['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.tolist()
    PERIODS0 = demand_data.index.tolist()
    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])
    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')
```

(continues on next page)

(continued from previous page)

```
    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:', redundancy.get_value())
        print('Cost:', 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))

    m.set_objective(cost, sense=so.MIN, name='cost_obj')
    res = m.solve()
    if res is not None:
        print('Redundancy:', redundancy.get_value())
        print('Cost:', 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.
```

(continues on next page)

(continued from previous page)

```
NOTE: Added action set 'optimization'.
NOTE: Converting model manpower_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 63 variables (0 free, 3 fixed).
NOTE: The problem has 24 linear constraints (6 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 108 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.03 seconds.
NOTE: The LP presolver removed 37 variables and 12 constraints.
NOTE: The LP presolver removed 53 constraint coefficients.
NOTE: The presolved problem has 26 variables, 12 constraints, and 55 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    -1.032250E+03         0
          P 2         16     8.417969E+02         0
NOTE: Optimal.
NOTE: Objective = 841.796875.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 24 rows and 4 columns.
Redundancy: 841.796875
Cost: 1576206.140351
               numWorkers numRecruits numRedundant numShortTime numExcess
1            2
semiskilled 0  1500.00000           -            -            -         -
semiskilled 1  1442.96875           0            0           50   17.9688
semiskilled 2  2000.00000         800            0            0         0
semiskilled 3  2525.00000     708.224            0           50         0
skilled     0  1000.00000           -            -            -         -
skilled     1  1025.00000           0            0           50         0
skilled     2  1500.00000         500            0            0         0
skilled     3  2000.00000         500            0            0         0
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  309.895833
semiskilled skilled      3  131.578947
unskilled   semiskilled 1  200.000000
unskilled   semiskilled 2  200.000000
unskilled   semiskilled 3  200.000000
                           numDowngrade
1            2           3
```

(continues on next page)

(continued from previous page)

```
semiskilled unskilled   1  0.000000e+00
semiskilled unskilled   2  0.000000e+00
semiskilled unskilled   3  0.000000e+00
skilled     semiskilled 1  1.684375e+02
skilled     semiskilled 2  2.181510e+02
skilled     semiskilled 3  0.000000e+00
skilled     unskilled   1  0.000000e+00
skilled     unskilled   2  0.000000e+00
skilled     unskilled   3 -1.421085e-14
NOTE: Added action set 'optimization'.
NOTE: Converting model manpower_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 63 variables (0 free, 3 fixed).
NOTE: The problem has 24 linear constraints (6 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 108 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 38 variables and 13 constraints.
NOTE: The LP presolver removed 56 constraint coefficients.
NOTE: The presolved problem has 25 variables, 11 constraints, and 52 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    -4.018114E+04         0
          D 2          6     4.986773E+05         0
NOTE: Optimal.
NOTE: Objective = 498677.28532.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 24 rows and 4 columns.
Redundancy: 1423.718837
Cost: 498677.285319
               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
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
```

(continues on next page)

(continued from previous page)

```
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  2.500000e+01
semiskilled unskilled    2  2.842171e-14
semiskilled unskilled    3 -2.842171e-14
skilled     semiskilled 1  0.000000e+00
skilled     semiskilled 2  0.000000e+00
skilled     semiskilled 3  0.000000e+00
skilled     unskilled    1  0.000000e+00
skilled     unskilled    2  1.136868e-13
skilled     unskilled    3  1.421085e-14
Out[2]: 498677.285319
```

7.1.6 Refinery Optimization

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex6_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex06.html

Model

```python
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],
```

(continues on next page)

(continued from previous page)

```
        ['crude1', 'residuum', 0.13],
        ['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'])
```

(continues on next page)

(continued from previous page)

```
    fuel_oil_ratio_data = pd.DataFrame([
        ['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', lb=0)
    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', lb=0)
    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')
```

(continues on next page)

(continued from previous page)

```
    octane = octane_data['octane']
    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')

    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()
                                  for a in ARCS if a[1] == p))
```

(continues on next page)

(continued from previous page)

```
        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: Added action set 'optimization'.
NOTE: Converting model refinery_optimization to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 51 variables (0 free, 0 fixed).
NOTE: The problem has 46 linear constraints (4 LE, 38 EQ, 3 GE, 1 range).
NOTE: The problem has 158 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
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    9.878656E+05         0
          P 2         18    2.113651E+05         0
NOTE: Optimal.
NOTE: Objective = 211365.13477.
NOTE: The Dual Simplex solve time is 0.01 seconds.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 51 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 46 rows and 4 columns.
        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
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
```

(continues on next page)

(continued from previous page)

```
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]: 211365.134769
```

7.1.7 Mining Optimization

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex7_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex07.html

Model

```python
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: Added action set 'optimization'.
NOTE: Converting model mining_optimization to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 60 variables (0 free, 0 fixed).
NOTE: The problem has 40 binary and 0 integer variables.
NOTE: The problem has 66 linear constraints (61 LE, 5 EQ, 0 GE, 0 range).
NOTE: The problem has 151 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 7 variables and 7 constraints.
NOTE: The MILP presolver removed 14 constraint coefficients.
NOTE: The MILP presolver modified 8 constraint coefficients.
NOTE: The presolved problem has 53 variables, 59 constraints, and 137 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     12     95.6438817    367.7788994   73.99%       0
                0        1     12     95.6438817    157.7309278   39.36%       0
                0        1     12     95.6438817    150.9548680   36.64%       0
                0        1     12     95.6438817    146.8619912   34.87%       0
                0        1     13    140.0319422    146.8619912    4.65%       0
NOTE: The MILP solver added 4 cuts with 16 cut coefficients at the root.
                1        1     14    146.8619808    146.8619912    0.00%       0
NOTE: Optimal within relative gap.
NOTE: Objective = 146.86198078.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 60 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 66 rows and 4 columns.
           isOpen  isWorked   extract
1      2
mine1 1  1.000000  1.000000  2.000000
mine1 2  1.000000  0.000000  0.000000
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
```

(continues on next page)

(continued from previous page)

```
mine2 2  1.000000  1.000000  2.500000
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  1.300000
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  2.200000
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            6.000000          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]: 146.861981
```

7.1.8 Farm Planning

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex8_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex08.html

Model

```python
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,
```

(continues on next page)

(continued from previous page)

```
                '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,
                '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']
```

(continues on next page)

(continued from previous page)

```
    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')
    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:
```

(continues on next page)

(continued from previous page)

```
            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}

    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')
```

(continues on next page)

(continued from previous page)

```
    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] <=
        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: Added action set 'optimization'.
```

(continues on next page)

(continued from previous page)

```
NOTE: Converting model farm_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 143 variables (0 free, 13 fixed).
NOTE: The problem has 101 linear constraints (25 LE, 70 EQ, 5 GE, 1 range).
NOTE: The problem has 780 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 86 variables and 71 constraints.
NOTE: The LP presolver removed 551 constraint coefficients.
NOTE: The presolved problem has 57 variables, 30 constraints, and 229 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    1.971960E+05         0
          D 2         55    1.217192E+05         0
NOTE: Optimal.
NOTE: Objective = 121719.17286.
NOTE: The Dual Simplex solve time is 0.03 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 143 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 101 rows and 4␣
↪columns.
        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
```

(continues on next page)

(continued from previous page)

```
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
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       revenue␣
↪          cost         profit
1                                                                                   ␣
↪
1        53.735000       30.935000            0.0                   0.0  41494.530000␣
↪ 19588.466667  21906.063333
2        52.341850       40.757423            0.0                   0.0  41153.336497␣
↪ 19264.639818  21888.696679
3        57.435807       57.435807            0.0                   0.0  45212.490308␣
↪ 19396.435208  25816.055100
4        56.964286       56.964286            0.0                   0.0  45860.056078␣
↪ 19034.285714  26825.770363
5        50.853436       50.853436            0.0                   0.0  42716.941438␣
↪ 17434.354053  25282.587385
          grainAcres  grainGrown
1       2
group1 1   20.000000   22.000000
```

(continues on next page)

(continued from previous page)

```
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
group4 3    0.000000    0.000000
group4 4    0.000000    0.000000
group4 5    0.000000    0.000000
   grainBought  grainSold  sugarBeetAcres  sugerBeetGrown  sugarBeetBought ␣
↪sugarBeetSold
1                                                                              ␣
↪
1    36.620000        0.0       60.766667       91.150000              0.0      22.
↪760000
2    35.100200        0.0       62.670049       94.005073              0.0      27.
↪388173
3    37.836507        0.0       65.100304       97.650456              0.0      24.
↪550338
4    40.142857        0.0       76.428571      114.642857              0.0      42.
↪142857
5    33.476475        0.0       87.539208      131.308812              0.0      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]: 121719.172861
```

7.1.9 Economic Planning

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex9_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex09.html

Model

```
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'])

    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, sort=True))
```

(continues on next page)

(continued from previous page)

```
    final_demand = so.get_solution_table(
        static_production, sort=True)['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
                                    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])
```

(continues on next page)

(continued from previous page)

```
    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, sort=True))
    print(so.get_solution_table(manpower_con.get_expressions(), sort=True))

    # 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:
            continuity_con[i, year].set_rhs(0)
    problem2.solve()
    print(so.get_solution_table(production, stock, extra_capacity,
                                productive_capacity, sort=True))
    print(so.get_solution_table(manpower_con.get_expressions(), sort=True))

    # 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, sort=True))
    print(so.get_solution_table(manpower_con.get_expressions(), sort=True))

    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: Added action set 'optimization'.
NOTE: Converting model economic_planning to OPTMODEL.
```

(continues on next page)

(continued from previous page)

```
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 9 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 0.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 3 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 3 rows and 4 columns.
           static_production
1
coal              166.396761
steel             105.668016
transport          92.307692
NOTE: Initialized model Problem1.
NOTE: Added action set 'optimization'.
NOTE: Converting model Problem1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 63 variables (0 free, 12 fixed).
NOTE: The problem has 42 linear constraints (24 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 255 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 18 variables and 3 constraints.
NOTE: The LP presolver removed 37 constraint coefficients.
NOTE: The presolved problem has 45 variables, 39 constraints, and 218 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    2.683246E+04         0
          P 2         42    2.141875E+03         0
NOTE: Optimal.
NOTE: Objective = 2141.8751967.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 42 rows and 4 columns.
            production        stock extra_capacity productive_capacity
```

(continues on next page)

(continued from previous page)

```
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 1.42109e-14             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 -1.45661e-13            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 -3.90799e-14            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
NOTE: Initialized model Problem2.
NOTE: Added action set 'optimization'.
NOTE: Converting model Problem2 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 63 variables (0 free, 12 fixed).
NOTE: The problem has 42 linear constraints (24 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 255 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 18 variables and 3 constraints.
NOTE: The LP presolver removed 37 constraint coefficients.
NOTE: The presolved problem has 45 variables, 39 constraints, and 218 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.504360E+04         0
          P 2         46    2.618579E+03         0
NOTE: Optimal.
NOTE: Objective = 2618.5791147.
```

(continues on next page)

(continued from previous page)

```
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 42 rows and 4 columns.
            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   -1.20732e-12  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
1
1    217.374162
2    344.581624
3    384.165212
4    470.000000
5    470.000000
6    150.000000
NOTE: Initialized model Problem3.
NOTE: Added action set 'optimization'.
NOTE: Converting model Problem3 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 63 variables (0 free, 12 fixed).
NOTE: The problem has 36 linear constraints (18 LE, 18 EQ, 0 GE, 0 range).
NOTE: The problem has 219 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
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.
```

(continues on next page)

(continued from previous page)

```
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.464016E+05         0
          D 2         54    2.450706E+03         0
          P 2         56    2.450027E+03         0
NOTE: Optimal.
NOTE: Objective = 2450.0266228.
NOTE: The Dual Simplex solve time is 0.03 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 36 rows and 4 columns.
            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
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]: 2450.026623
```

7.1.10 Decentralization

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex10_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex10.html

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(
            ['i', 'j'])

    max_num_depts = 3

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

(continues on next page)

(continued from previous page)

```
                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, name='product')

    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='pd1')

    product_def2 = m.add_constraints((product[i, j, k, l] <= assign[i, j]
                                      for (i, j, k, l) in IJKL),
                                     name='pd2')

    product_def3 = m.add_constraints((product[i, j, k, l] <= assign[k, l]
                                      for (i, j, k, l) in IJKL),
                                     name='pd3')

    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]
```

(continues on next page)

(continued from previous page)

```
        for i in DEPTS for k in DEPTS for l in CITIES if i < k),
        name='pd4')

    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='pd5')

    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: Added action set 'optimization'.
NOTE: Converting model decentralization to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 105 variables (0 free, 0 fixed).
NOTE: The problem has 105 binary and 0 integer variables.
NOTE: The problem has 278 linear constraints (183 LE, 5 EQ, 90 GE, 0 range).
NOTE: The problem has 660 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
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 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      2    -14.9000000    135.0000000  111.04%       0
                0        1      2    -14.9000000     67.5000000  122.07%       0
                0        1      2    -14.9000000     55.0000000  127.09%       0
                0        1      3      8.1000000     55.0000000   85.27%       0
                0        1      3      8.1000000     48.0000000   83.12%       0
                0        1      3      8.1000000     44.8375000   81.93%       0
                0        1      3      8.1000000     42.0000000   80.71%       0
                0        1      3      8.1000000     39.0666667   79.27%       0
                0        1      3      8.1000000     34.7500000   76.69%       1
```

(continues on next page)

(continued from previous page)

```
                0        1      3      8.1000000     33.3692308   75.73%       1
                0        1      3      8.1000000     32.6500000   75.19%       1
                0        1      3      8.1000000     31.9066667   74.61%       1
                0        1      3      8.1000000     30.7000000   73.62%       1
                0        1      3      8.1000000     30.1600000   73.14%       1
                0        1      3      8.1000000     29.8800000   72.89%       1
                0        1      3      8.1000000     29.8000000   72.82%       1
                0        1      3      8.1000000     29.4722222   72.52%       1
                0        1      3      8.1000000     28.9117647   71.98%       1
                0        1      3      8.1000000     28.6716667   71.75%       1
                0        1      3      8.1000000     28.5000000   71.58%       2
                0        1      4     14.9000000     14.9000000    0.00%       2
NOTE: The MILP solver added 34 cuts with 185 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 105 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 278 rows and 4␣
↪columns.
Problem Summary

                                Value
Label
Objective Sense           Maximization
Objective Function          netBenefit
Objective Type                  Linear

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

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

Constraint Coefficients            660
NOTE: Added action set 'optimization'.
NOTE: Converting model decentralization to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 105 variables (0 free, 0 fixed).
NOTE: The problem has 105 binary and 0 integer variables.
NOTE: The problem has 68 linear constraints (3 LE, 65 EQ, 0 GE, 0 range).
NOTE: The problem has 270 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
```

(continues on next page)

(continued from previous page)

```
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 32 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      2    -28.1000000    135.0000000  120.81%       0
                0        1      2    -28.1000000     30.0000000  193.67%       0
                0        1      3    -16.3000000     30.0000000  154.33%       0
                0        1      3    -16.3000000     30.0000000  154.33%       0
NOTE: The MILP solver added 4 cuts with 24 cut coefficients at the root.
                2        0      4     14.9000000     14.9000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 105 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 68 rows and 4 columns.
Problem Summary

                                Value
Label
Objective Sense           Maximization
Objective Function          netBenefit
Objective Type                  Linear

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

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

(continues on next page)

(continued from previous page)

```
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
```

7.1.11 Optimal Wedding

Reference

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

Model

```
import sasoptpy as so
import math


def test(cas_conn, num_guests=20, 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(options={
        'with': 'milp', 'decomp': {'method': 'set'}, 'presolver': 'none'})

    if res is not None:

        print(so.get_solution_table(x))
```

(continues on next page)

(continued from previous page)

```
        # 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)
NOTE: Initialized model wedding.
NOTE: Added action set 'optimization'.
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 TMP4V46PYQN␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMP4V46PYQN has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: The problem wedding has 147 variables (140 binary, 0 integer, 0 free, 0 fixed).
NOTE: The problem has 1357 constraints (7 LE, 20 EQ, 1330 GE, 0 range).
NOTE: The problem has 4270 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: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster␣
↪branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 7 blocks. The largest block␣
↪covers 14.08% of the constraints in the problem.
NOTE: The decomposition subproblems cover 147 (100%) variables and 1337 (98.53%)␣
↪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      13.0000      13.0000 1.30e+01 1.30e+01    0    0
            1       0.0000      13.0000      13.0000 1.30e+01 1.30e+01    1    1
            .       0.0000      13.0000      13.0000 1.30e+01 1.30e+01    7   11
           10       0.0000      13.0000      13.0000 1.30e+01 1.30e+01    8   12
           18       4.2500      13.0000      13.0000  205.88%  205.88%   22   30
           19       6.0000      13.0000      13.0000  116.67%  116.67%   23   32
            .       6.0000      13.0000      13.0000  116.67%  116.67%   23   32
           20       6.0000      13.0000      13.0000  116.67%  116.67%   24   33
           21       9.5000      13.0000      13.0000   36.84%   36.84%   25   35
```

(continues on next page)

(continued from previous page)

```
           23      13.0000      13.0000      13.0000    0.00%    0.00%   27   37
            Node  Active   Sols         Best         Best      Gap    CPU   Real
                                     Integer        Bound            Time   Time
               0       1      3      13.0000      13.0000    0.00%     27     37
NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 37.98 seconds.
NOTE: Optimal.
NOTE: Objective = 13.
        x
1  2
1  1  1.0
1  2  0.0
1  3  0.0
1  4  0.0
1  5  0.0
1  6  0.0
1  7  0.0
2  1  1.0
2  2  0.0
2  3  0.0
2  4  0.0
2  5  0.0
2  6  0.0
2  7  0.0
3  1  1.0
3  2  0.0
3  3  0.0
3  4  0.0
3  5  0.0
3  6  0.0
3  7  0.0
4  1  0.0
4  2  1.0
4  3  0.0
4  4  0.0
4  5  0.0
4  6  0.0
4  7  0.0
5  1  0.0
5  2  1.0
...   ...
16 6  1.0
16 7  0.0
17 1  0.0
17 2  0.0
17 3  0.0
17 4  0.0
17 5  0.0
17 6  1.0
17 7  0.0
18 1  0.0
18 2  0.0
18 3  0.0
18 4  0.0
18 5  0.0
18 6  1.0
18 7  0.0
```

(continues on next page)

(continued from previous page)

```
19 1  0.0
19 2  0.0
19 3  0.0
19 4  0.0
19 5  0.0
19 6  0.0
19 7  1.0
20 1  0.0
20 2  0.0
20 3  0.0
20 4  0.0
20 5  0.0
20 6  0.0
20 7  1.0

[140 rows x 1 columns]
Table 1 : [ 1 2 3 ]
Table 2 : [ 4 5 6 ]
Table 3 : [ 7 8 9 ]
Table 4 : [ 10 11 12 ]
Table 5 : [ 13 14 15 ]
Table 6 : [ 16 17 18 ]
Table 7 : [ 19 20 ]
Out[2]: 13.0
```

7.1.12 Kidney Exchange

Reference

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

    # Model
```

(continues on next page)

(continued from previous page)

```
    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(options={'with': '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(verbose=True, options={
        'with': '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: Added action set 'optimization'.
NOTE: Converting model kidney_exchange to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 8133 variables (0 free, 0 fixed).
NOTE: The problem has 8133 binary and 0 integer variables.
NOTE: The problem has 5967 linear constraints (38 LE, 5929 EQ, 0 GE, 0 range).
NOTE: The problem has 24245 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The remaining solution time after problem generation and solver initialization␣
↪is 299.61 seconds.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 6193 variables and 5357 constraints.
NOTE: The MILP presolver removed 17478 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 1940 variables, 610 constraints, and 6767 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 80 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      4     12.0775275   2279.8770216   99.47%       1
                0        1      4     12.0775275     18.3085704   34.03%       1
NOTE: The MILP solver's symmetry detection found 140 orbits. The largest orbit␣
↪contains 37 variables.
                0        1      4     12.0775275     18.3085704   34.03%       2
                0        1      4     12.0775275     18.3085704   34.03%       2
                0        1      4     12.0775275     18.3085704   34.03%       3
                0        1      4     12.0775275     18.3085704   34.03%       3
NOTE: The MILP solver added 4 cuts with 196 cut coefficients at the root.
               11        8      5     17.1113590     18.1252274    5.59%       4
               23        4      6     17.1113590     18.0210902    5.05%       4
               27        3      6     17.1113590     18.0210902    5.05%       5
               29        3      7     17.1113590     17.9798537    4.83%       5
               67       11      8     17.1113590     17.7951852    3.84%       6
               73        9      9     17.1113590     17.7951852    3.84%       6
               74        8     10     17.1113590     17.7951852    3.84%       6
              176        0     10     17.1113590     17.1113590    0.00%       7
NOTE: Optimal.
NOTE: Objective = 17.111358985.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 8133 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 5967 rows and 4␣
↪columns.
NOTE: Added action set 'optimization'.
NOTE: Converting model kidney_exchange to DataFrame.
NOTE: Cloud Analytic Services made the uploaded file available as table BLOCKSTABLE␣
↪in caslib CASUSERHDFS(casuser).
```

(continues on next page)

(continued from previous page)

```
NOTE: The table BLOCKSTABLE has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPWRTEVWO4␣
↪in caslib CASUSERHDFS(casuser).
NOTE: The table TMPWRTEVWO4 has been created in caslib CASUSERHDFS(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
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 299.39 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.
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: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster␣
↪branching.
NOTE: The number of block threads has been reduced to 1 threads.
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      10.5814      10.5814   96.27%    96.27%    2    2
            1     259.0121      10.5814      10.5814   95.91%    95.91%    3    4
            2     230.6758      10.5814      10.5814   95.41%    95.41%    4    5
            3     204.2627      10.5814      10.5814   94.82%    94.82%    5    6
            4     192.9770      14.7394      14.7394   92.36%    92.36%    6    9
            6     162.6582      15.6274      15.6274   90.39%    90.39%   10   13
            7     140.2584      15.6274      15.6274   88.86%    88.86%   11   15
            8     110.8604      15.6274      15.6274   85.90%    85.90%   13   17
            9     108.1582      15.8122      15.6274   85.38%    85.55%   14   18
           10     108.1582      15.8122      17.1114   85.38%    84.18%   14   19
           11      46.6890      17.1114      17.1114   63.35%    63.35%   15   20
           12      39.3063      17.1114      17.1114   56.47%    56.47%   17   22
           13      17.1114      17.1114      17.1114    0.00%     0.00%   18   24
            Node  Active   Sols         Best         Best      Gap     CPU    Real
                                     Integer        Bound             Time    Time
               0       1      9      17.1114      17.1114    0.00%      18      24
NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 24.75 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 17.111358985.
Out[2]: 17.111359
```

7.1.13 Multiobjective

Reference

https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=ormpug&docsetTarget=ormpug_lsosolver_examples07.htm&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/lsoe10.html

Model

```
import sasoptpy as so


def test(cas_conn, sols=False):

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

    x = m.add_variables([1, 2], lb=0, ub=5, name='x')

    f1 = m.set_objective((x[1]-1)**2 + (x[1] - x[2])**2, name='f1', sense=so.MIN,␣
↪multiobj=True)
    f2 = m.set_objective((x[1]-x[2])**2 + (x[2] - 3)**2, name='f2', sense=so.MIN,␣
↪multiobj=True)

    m.solve(verbose=True, options={'with': 'lso', 'obj': (f1, f2), 'logfreq': 50})

    print('f1', f1.get_value())
    print('f2', f2.get_value())

    if sols:
        return cas_conn.CASTable('allsols').to_frame()
    else:
        return f1.get_value()
```

Output

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

In [2]: sols = test(cas_conn, sols=True)
NOTE: Initialized model multiobjective.
NOTE: Added action set 'optimization'.
NOTE: Converting model multiobjective to OPTMODEL.
var x {{1,2}} >= 0 <= 5;
min f1 = (x[1] - 1) ^ (2) + (x[1] - x[2]) ^ (2);
min f2 = (x[1] - x[2]) ^ (2) + (x[2] - 3) ^ (2);
solve with lso obj (f1 f2) / logfreq=50;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
create data allsols from [s]=(1.._NVAR_) name=_VAR_[s].name {j in 1.._NSOL_} <col(
↪'sol_'||j)=_VAR_[s].sol[j]>;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
```

(continues on next page)

(continued from previous page)

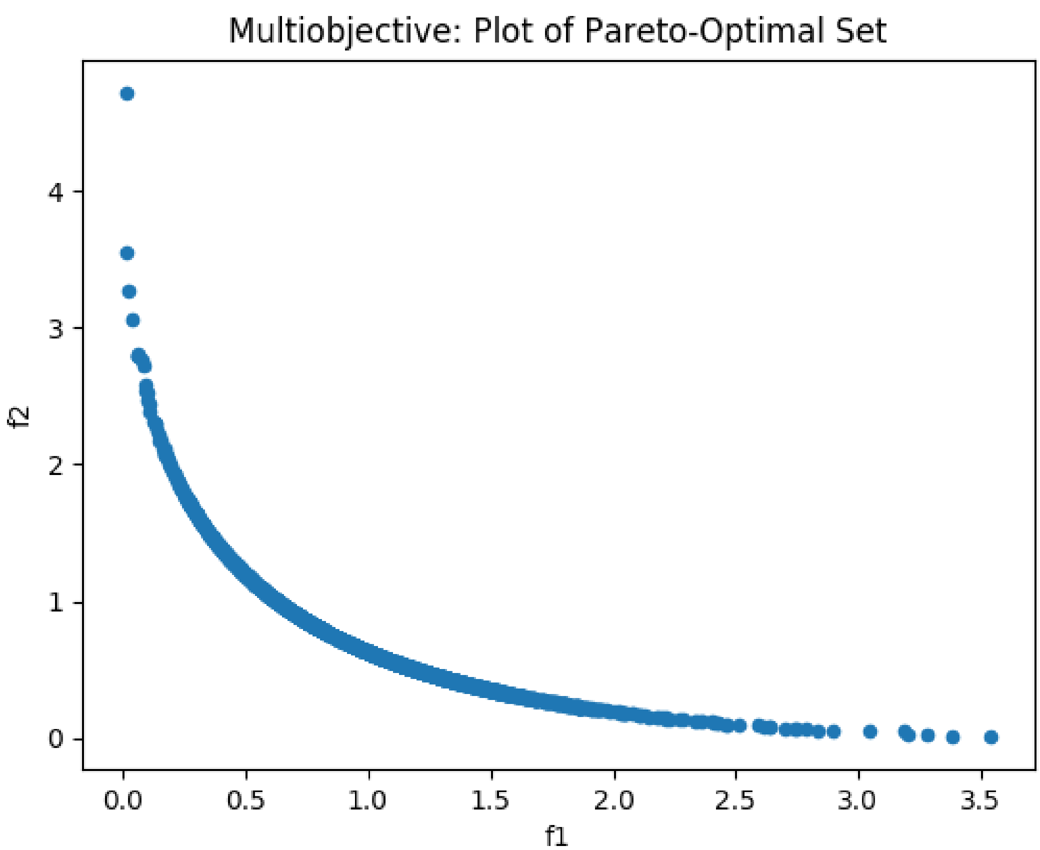
```
NOTE: The problem has 2 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0␣
↪nonlinear constraints.
NOTE: The SOLVE statement is executing in the distributed computing environment with␣
↪140 worker nodes.
NOTE: The LSO solver is using up to 5872 total threads across 140 machines.
NOTE: The LSO solver is using the EAGLS optimizer algorithm.
NOTE: The problem has 2 variables (0 integer, 2 continuous).
NOTE: The problem has 0 constraints (0 linear, 0 nonlinear).
NOTE: The problem has 2 user-defined functions.
NOTE: The deterministic parallel mode is enabled.
           Iteration    Nondom         Progress    Infeasibility     Evals      Time
                   1         4                .                0        84         0
                  51       877        0.0000811                0      2876         3
                 101      1654        0.0000123                0      5576         6
                 151      2290   0.000003230545                0      8181        10
                 201      2874        0.0000182                0     10796        13
                 251      3422   0.000003148003                0     13447        17
                 301      3847   0.000001559509                0     16046        22
                 351      4282   0.000001159917                0     18733        26
                 401      4712   0.000002423148                0     21315        31
                 428      4932   0.000000704951                0     22734        34
NOTE: Function convergence criteria reached.
NOTE: The output table 'ALLSOLS' in caslib 'CASUSERHDFS(casuser)' has 2 rows and 4934␣
↪columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 14 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 12 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 2 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 0 rows and 4 columns.
f1 0.014971
f2 4.721632
```

```
In [3]: import matplotlib.pyplot as plt

In [4]: tr = sols.transpose()

In [5]: scvalues = tr.iloc[2:]

In [6]: scvalues = scvalues.astype({0: float, 1: float})

In [7]: x = sasoptpy.get_obj_by_name('x')

In [8]: f1 = sasoptpy.get_obj_by_name('f1')

In [9]: f2 = sasoptpy.get_obj_by_name('f2')

In [10]: x[1].set_value(scvalues[0])

In [11]: x[2].set_value(scvalues[1])

In [12]: scvalues['f1'] = f1.get_value()
```

(continues on next page)

(continued from previous page)

```
In [13]: scvalues['f2'] = f2.get_value()

In [14]: f = scvalues.plot.scatter(x='f1', y='f2')

In [15]: f.set_title('Multiobjective: Plot of Pareto-Optimal Set');

In [16]: f
Out[16]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddddaa0198>
```



7.2 Viya Examples / Abstract

7.2.1 Curve Fitting

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex11_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex11.html

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn, sols=False):

    # Upload data to server first
    xy_raw = pd.DataFrame([
        [0.0, 1.0],
        [0.5, 0.9],
        [1.0, 0.7],
        [1.5, 1.5],
        [1.9, 2.0],
        [2.5, 2.4],
        [3.0, 3.2],
        [3.5, 2.0],
        [4.0, 2.7],
        [4.5, 3.5],
        [5.0, 1.0],
        [5.5, 4.0],
        [6.0, 3.6],
        [6.6, 2.7],
        [7.0, 5.7],
        [7.6, 4.6],
        [8.5, 6.0],
        [9.0, 6.8],
        [10.0, 7.3]
        ], columns=['x', 'y'])
    xy_data = cas_conn.upload_frame(xy_raw, casout={'name': 'xy_data',
                                                    'replace': True})

    # Read observations
    POINTS, (x, y), xy_table_ref = so.read_table(xy_data, columns=['x', 'y'])

    # Parameters and variables
    order = so.Parameter(name='order')
    beta = so.VariableGroup(so.exp_range(0, order), name='beta')
    estimate = so.ImplicitVar(
        (beta[0] + so.quick_sum(beta[k] * x[i] ** k
                                for k in so.exp_range(1, order))
         for i in POINTS), name='estimate')

    surplus = so.VariableGroup(POINTS, name='surplus', lb=0)
    slack = so.VariableGroup(POINTS, name='slack', lb=0)

    objective1 = so.Expression(
        so.quick_sum(surplus[i] + slack[i] for i in POINTS), name='objective1')
    abs_dev_con = so.ConstraintGroup(
        (estimate[i] - surplus[i] + slack[i] == y[i] for i in POINTS),
        name='abs_dev_con')

    minmax = so.Variable(name='minmax')
    objective2 = so.Expression(minmax + 0.0, name='objective2')
    minmax_con = so.ConstraintGroup(
        (minmax >= surplus[i] + slack[i] for i in POINTS), name='minmax_con')
```

(continues on next page)

(continued from previous page)

```
    order.set_init(1)
    L1 = so.Model(name='L1', session=cas_conn)
    L1.set_objective(objective1, sense=so.MIN)
    L1.include(POINTS, x, y, xy_table_ref)
    L1.include(order, beta, estimate, surplus, slack, abs_dev_con)
    L1.add_statement('print x y estimate surplus slack;', after_solve=True)

    L1.solve(verbose=True)
    sol_data1 = L1.response['Print3.PrintTable'].sort_values('x')
    print(so.get_solution_table(beta))
    print(sol_data1.to_string())

    Linf = so.Model(name='Linf', session=cas_conn)
    Linf.include(L1, minmax, minmax_con)
    Linf.set_objective(objective2, sense=so.MIN)

    Linf.solve()
    sol_data2 = Linf.response['Print3.PrintTable'].sort_values('x')
    print(so.get_solution_table(beta))
    print(sol_data2.to_string())

    order.set_init(2)

    L1.solve()
    sol_data3 = L1.response['Print3.PrintTable'].sort_values('x')
    print(so.get_solution_table(beta))
    print(sol_data3.to_string())

    Linf.solve()
    sol_data4 = Linf.response['Print3.PrintTable'].sort_values('x')
    print(so.get_solution_table(beta))
    print(sol_data4.to_string())

    if sols:
        return (sol_data1, sol_data2, sol_data3, sol_data4)
    else:
        return Linf.get_objective_value()
```

Output

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

In [2]: (s1, s2, s3, s4) = test(cas_conn, sols=True)
NOTE: Cloud Analytic Services made the uploaded file available as table XY_DATA in
↪caslib CASUSERHDFS(casuser).
NOTE: The table XY_DATA has been created in caslib CASUSERHDFS(casuser) from binary
↪data uploaded to Cloud Analytic Services.
NOTE: Initialized model L1.
NOTE: Added action set 'optimization'.
NOTE: Converting model L1 to OPTMODEL.
set set_XY_DATA_N;
num x {set_XY_DATA_N};
num y {set_XY_DATA_N};
read data XY_DATA into set_XY_DATA_N=[_N_] x y;
```

(continues on next page)

(continued from previous page)

```
num order init 1;
var beta {{0..order}};
impvar estimate {i_1 in set_XY_DATA_N} = beta[0] + sum {i_2 in 1..order}(beta[i_2] *␣
↪(x[i_1]) ^ (i_2));
var surplus {{set_XY_DATA_N}} >= 0;
var slack {{set_XY_DATA_N}} >= 0;
min objective1 = sum {i_3 in set_XY_DATA_N}(surplus[i_3] + slack[i_3]);
con abs_dev_con {i_4 in set_XY_DATA_N} : y[i_4] - estimate[i_4] + surplus[i_4] -␣
↪slack[i_4] = 0;

solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
print x y estimate surplus slack;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: There were 19 rows read from table 'XY_DATA' in caslib 'CASUSERHDFS(casuser)'.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 40 variables (2 free, 0 fixed).
NOTE: The problem uses 19 implicit variables.
NOTE: The problem has 19 linear constraints (0 LE, 19 EQ, 0 GE, 0 range).
NOTE: The problem has 75 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 38 variables and 0 constraints.
NOTE: The LP presolver removed 38 constraint coefficients.
NOTE: The LP presolver formulated the dual of the problem.
NOTE: The presolved problem has 19 variables, 2 constraints, and 37 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.160000E+01         0
          D 2          5    1.146625E+01         0
NOTE: Optimal.
NOTE: Objective = 11.46625.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 40 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 19 rows and 4 columns.
      beta
1
0  0.58125
1  0.63750
    COL1     x    y  estimate       surplus    slack
10  11.0   0.0  1.0   0.58125  0.000000e+00  0.41875
18  19.0   0.5  0.9   0.90000  5.551115e-17  0.00000
12  13.0   1.0  0.7   1.21875  5.187500e-01  0.00000
4    5.0   1.5  1.5   1.53750  3.750000e-02  0.00000
0    1.0   1.9  2.0   1.79250  0.000000e+00  0.20750
```

(continues on next page)

(continued from previous page)

```
15  16.0   2.5  2.4   2.17500  0.000000e+00  0.22500
1    2.0   3.0  3.2   2.49375  0.000000e+00  0.70625
11  12.0   3.5  2.0   2.81250  8.125000e-01  0.00000
5    6.0   4.0  2.7   3.13125  4.312500e-01  0.00000
3    4.0   4.5  3.5   3.45000  0.000000e+00  0.05000
8    9.0   5.0  1.0   3.76875  2.768750e+00  0.00000
14  15.0   5.5  4.0   4.08750  8.750000e-02  0.00000
13  14.0   6.0  3.6   4.40625  8.062500e-01  0.00000
7    8.0   6.6  2.7   4.78875  2.088750e+00  0.00000
9   10.0   7.0  5.7   5.04375  0.000000e+00  0.65625
17  18.0   7.6  4.6   5.42625  8.262500e-01  0.00000
2    3.0   8.5  6.0   6.00000  0.000000e+00  0.00000
6    7.0   9.0  6.8   6.31875  0.000000e+00  0.48125
16  17.0  10.0  7.3   6.95625  0.000000e+00  0.34375
NOTE: Initialized model Linf.
NOTE: Added action set 'optimization'.
NOTE: Converting model Linf to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: There were 19 rows read from table 'XY_DATA' in caslib 'CASUSERHDFS(casuser)'.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 41 variables (3 free, 0 fixed).
NOTE: The problem uses 19 implicit variables.
NOTE: The problem has 38 linear constraints (0 LE, 19 EQ, 19 GE, 0 range).
NOTE: The problem has 132 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 41 variables, 38 constraints, and 132 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    -1.900000E+00         0
          P 2         26     1.725000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.725.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 41 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 38 rows and 4 columns.
    beta
1
0 -0.400
1  0.625
    COL1     x    y  estimate  surplus   slack
10  11.0   0.0  1.0   -0.4000    0.000  1.4000
18  19.0   0.5  0.9   -0.0875    0.000  0.9875
12  13.0   1.0  0.7    0.2250    0.000  0.4750
4    5.0   1.5  1.5    0.5375    0.000  0.9625
```

(continues on next page)

(continued from previous page)

```
0    1.0   1.9  2.0    0.7875    0.000  1.2125
15  16.0   2.5  2.4    1.1625    0.000  1.2375
1    2.0   3.0  3.2    1.4750    0.000  1.7250
11  12.0   3.5  2.0    1.7875    0.000  0.2125
5    6.0   4.0  2.7    2.1000    0.000  0.6000
3    4.0   4.5  3.5    2.4125    0.000  1.0875
8    9.0   5.0  1.0    2.7250    1.725  0.0000
14  15.0   5.5  4.0    3.0375    0.000  0.9625
13  14.0   6.0  3.6    3.3500    0.000  0.2500
7    8.0   6.6  2.7    3.7250    1.025  0.0000
9   10.0   7.0  5.7    3.9750    0.000  1.7250
17  18.0   7.6  4.6    4.3500    0.000  0.2500
2    3.0   8.5  6.0    4.9125    0.000  1.0875
6    7.0   9.0  6.8    5.2250    0.000  1.5750
16  17.0  10.0  7.3    5.8500    0.000  1.4500
NOTE: Added action set 'optimization'.
NOTE: Converting model L1 to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: There were 19 rows read from table 'XY_DATA' in caslib 'CASUSERHDFS(casuser)'.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 41 variables (3 free, 0 fixed).
NOTE: The problem uses 19 implicit variables.
NOTE: The problem has 19 linear constraints (0 LE, 19 EQ, 0 GE, 0 range).
NOTE: The problem has 93 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 38 variables and 0 constraints.
NOTE: The LP presolver removed 38 constraint coefficients.
NOTE: The LP presolver formulated the dual of the problem.
NOTE: The presolved problem has 19 variables, 3 constraints, and 55 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.160000E+01         0
          D 2          5    1.045896E+01         0
NOTE: Optimal.
NOTE: Objective = 10.458964706.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 41 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 19 rows and 4 columns.
       beta
1
0  0.982353
1  0.294510
2  0.033725
   COL1     x    y  estimate       surplus     slack
10  11.0   0.0  1.0  0.982353  0.000000e+00  0.017647
18  19.0   0.5  0.9  1.138039  2.380392e-01  0.000000
```

(continues on next page)

(continued from previous page)

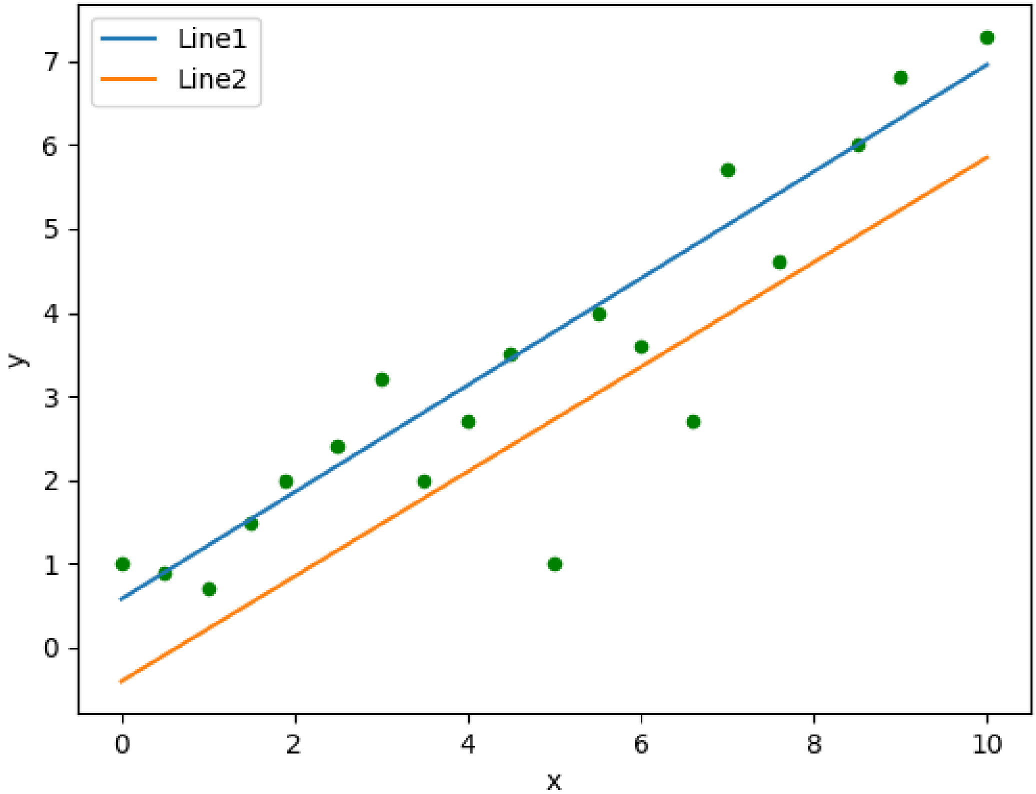
```
12  13.0   1.0  0.7  1.310588  6.105882e-01  0.000000
4    5.0   1.5  1.5  1.500000 -6.938894e-17  0.000000
0    1.0   1.9  2.0  1.663671  0.000000e+00  0.336329
15  16.0   2.5  2.4  1.929412  0.000000e+00  0.470588
1    2.0   3.0  3.2  2.169412  0.000000e+00  1.030588
11  12.0   3.5  2.0  2.426275  4.262745e-01  0.000000
5    6.0   4.0  2.7  2.700000 -1.110223e-16  0.000000
3    4.0   4.5  3.5  2.990588  0.000000e+00  0.509412
8    9.0   5.0  1.0  3.298039  2.298039e+00  0.000000
14  15.0   5.5  4.0  3.622353  0.000000e+00  0.377647
13  14.0   6.0  3.6  3.963529  3.635294e-01  0.000000
7    8.0   6.6  2.7  4.395200  1.695200e+00  0.000000
9   10.0   7.0  5.7  4.696471  0.000000e+00  1.003529
17  18.0   7.6  4.6  5.168612  5.686118e-01  0.000000
2    3.0   8.5  6.0  5.922353  0.000000e+00  0.077647
6    7.0   9.0  6.8  6.364706  0.000000e+00  0.435294
16  17.0  10.0  7.3  7.300000  4.440892e-16  0.000000
NOTE: Added action set 'optimization'.
NOTE: Converting model Linf to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: There were 19 rows read from table 'XY_DATA' in caslib 'CASUSERHDFS(casuser)'.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 42 variables (4 free, 0 fixed).
NOTE: The problem uses 19 implicit variables.
NOTE: The problem has 38 linear constraints (0 LE, 19 EQ, 19 GE, 0 range).
NOTE: The problem has 150 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver time is 0.00 seconds.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 42 variables, 38 constraints, and 150 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                             Objective
         Phase Iteration        Value         Time
          D 2          1   -1.900000E+00         0
          P 2         29    1.475000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.475.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 13 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 42 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 38 rows and 4 columns.
    beta
1
0  2.475
1 -0.625
2  0.125
    COL1     x    y  estimate   surplus     slack
10  11.0   0.0  1.0   2.47500  1.475000  0.000000
```

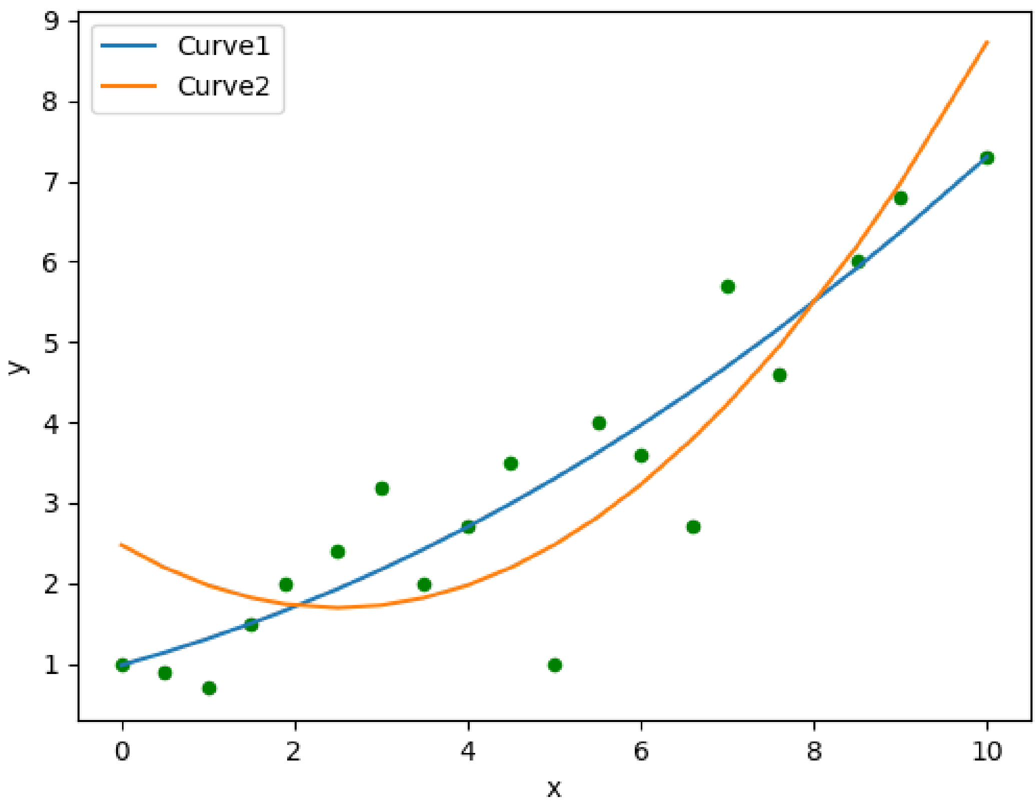
(continues on next page)

(continued from previous page)

```
18  19.0   0.5  0.9   2.19375  1.293750  0.000000
12  13.0   1.0  0.7   1.97500  1.275000  0.000000
4    5.0   1.5  1.5   1.81875  0.318750  0.000000
0    1.0   1.9  2.0   1.73875  0.606875  0.868125
15  16.0   2.5  2.4   1.69375  0.000000  0.706250
1    2.0   3.0  3.2   1.72500  0.000000  1.475000
11  12.0   3.5  2.0   1.81875  0.000000  0.181250
5    6.0   4.0  2.7   1.97500  0.000000  0.725000
3    4.0   4.5  3.5   2.19375  0.000000  1.306250
8    9.0   5.0  1.0   2.47500  1.475000  0.000000
14  15.0   5.5  4.0   2.81875  0.000000  1.181250
13  14.0   6.0  3.6   3.22500  0.000000  0.375000
7    8.0   6.6  2.7   3.79500  1.095000  0.000000
9   10.0   7.0  5.7   4.22500  0.000000  1.475000
17  18.0   7.6  4.6   4.94500  0.345000  0.000000
2    3.0   8.5  6.0   6.19375  0.193750  0.000000
6    7.0   9.0  6.8   6.97500  0.175000  0.000000
16  17.0  10.0  7.3   8.72500  1.425000  0.000000
```

```
# Plots
In [3]: import matplotlib.pyplot as plt

In [4]: p1 = s1.plot.scatter(x='x', y='y', c='g')

In [5]: s1.plot.line(ax=p1, x='x', y='estimate', label='Line1');

In [6]: s2.plot.line(ax=p1, x='x', y='estimate', label='Line2');

In [7]: p1
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddddc6ef60>

In [8]: p2 = s3.plot.scatter(x='x', y='y', c='g')

In [9]: s3.plot.line(ax=p2, x='x', y='estimate', label='Curve1');

In [10]: s4.plot.line(ax=p2, x='x', y='estimate', label='Curve2');

In [11]: p2
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7fddddad87f0>
```


Line1
Line2
y
x
0
1
2
3
4
5
6
7
0
2
4
6
8
10



7.2.2 Nonlinear 1

Reference

http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_nlpsolver_examples01.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/nlpse01.html

Model

```
import sasoptpy as so


def test(cas_conn):

    m = so.Model(name='nlpse01', session=cas_conn)
    x = m.add_variables(range(1, 9), lb=0.1, ub=10, name='x')

    f = so.Expression(0.4 * (x[1]/x[7]) ** 0.67 + 0.4 * (x[2]/x[8]) ** 0.67 + 10 -
↪x[1] - x[2], name='f')
    m.set_objective(f, sense=so.MIN)
```

(continues on next page)

(continued from previous page)

```
    m.add_constraint(1 - 0.0588*x[5]*x[7] - 0.1*x[1] >= 0, name='c1')
    m.add_constraint(1 - 0.0588*x[6]*x[8] - 0.1*x[1] - 0.1*x[2] >= 0, name='c2')
    m.add_constraint(1 - 4*x[3]/x[5] - 2/(x[3]**0.71 * x[5]) - 0.0588*(x[7]/x[3]**1.
↪3) >= 0, name='c3')
    m.add_constraint(1 - 4*x[4]/x[6] - 2/(x[4]**0.71 * x[6]) - 0.0588*(x[8]/x[4]**1.
↪3) >= 0, name='c4')
    m.add_constraint(f == [0.1, 4.2])

    x[1].set_init(6)
    x[2].set_init(3)
    x[3].set_init(0.4)
    x[4].set_init(0.2)
    x[5].set_init(6)
    x[6].set_init(6)
    x[7].set_init(1)
    x[8].set_init(0.5)

    m.add_statement('print x;', after_solve=True)
    m.solve(verbose=True, options={'with': 'nlp', 'algorithm': 'activeset'})
    print(m.get_problem_summary())
    print(m.get_solution_summary())
    if m.test_session() == 'CAS':
        print(m.response['Print3.PrintTable'])

    return m.get_objective_value()
```

Output

```
In [1]: from examples.nonlinear_1 import test

In [2]: test(cas_conn)
NOTE: Initialized model nlpse01.
NOTE: Added action set 'optimization'.
NOTE: Converting model nlpse01 to OPTMODEL.
var x {{1,2,3,4,5,6,7,8}} >= 0.1 <= 10;
x[1] = 6;
x[2] = 3;
x[3] = 0.4;
x[4] = 0.2;
x[5] = 6;
x[6] = 6;
x[7] = 1;
x[8] = 0.5;
min f = 0.4 * (((x[1]) / (x[7])) ^ (0.67)) + 0.4 * (((x[2]) / (x[8])) ^ (0.67)) -␣
↪x[1] - x[2] + 10.0;
con c1 : - 0.0588 * x[5] * x[7] - 0.1 * x[1] >= -1.0;
con c2 : - 0.0588 * x[6] * x[8] - 0.1 * x[1] - 0.1 * x[2] >= -1.0;
con c3 : - (4 * x[3]) / (x[5]) - (2) / ((x[3]) ^ (0.71) * x[5]) - 0.0588 * ((x[7]) /␣
↪((x[3]) ^ (1.3))) >= -1.0;
con c4 : - (4 * x[4]) / (x[6]) - (2) / ((x[4]) ^ (0.71) * x[6]) - 0.0588 * ((x[8]) /␣
↪((x[4]) ^ (1.3))) >= -1.0;
con con_1 : -9.9 <= 0.4 * (((x[1]) / (x[7])) ^ (0.67)) + 0.4 * (((x[2]) / (x[8])) ^␣
↪(0.67)) - x[1] - x[2] <= -5.8;
```

(continues on next page)

(continued from previous page)

```
solve with nlp / algorithm=activeset;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
print x;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 8 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 5 nonlinear constraints (0 LE, 0 EQ, 4 GE, 1 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0␣
↪nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Active Set algorithm is used.
                           Objective                              Optimality
              Iter              Value     Infeasibility                Error
                 0         3.65736570        0.41664483           0.24247905
                 1         3.65736570        0.41664483           0.24247905
                 2         3.40486061        0.10284726           0.10617988
                 3         3.51178229        0.07506389           0.10593173
                 4         4.23595983        0.03595983           0.33749510
                 5         4.16334906                 0           0.26471063
                 6         4.03168584        0.00791810           0.13742971
                 7         3.88912660        0.11248991           0.06129662
                 8         3.89579714        0.09534670           0.05994916
                 9         3.95046640        0.02649207           0.06776850
                10         3.92833580        0.03517161           0.06442935
                11         3.95179326        0.00494247           0.05837915
                12         3.94741555        0.00651989           0.05477333
                13         3.95209064        0.00058609           0.05265725
                14         3.95058104        0.00122758           0.04772557
                15         3.95055959        0.00099113           0.04613473
                16         3.95141460        0.00000381           0.04497006
                17         3.95132211   0.0000005999371           0.04260039
                18         3.95114031        0.00000941           0.04093117
                19         3.95027690        0.00011307           0.00020755
                20         3.95115797   0.0000007730235           0.00010507
                21         3.95116558                 0           0.00001366
                22         3.95116364   0.0000000153799           0.00000814
                23         3.95116355   0.0000000228326           0.00000595
                24         3.95116352   0.0000000257138           0.00000337
                25         3.95116349   0.0000000200547           0.00000132
                26         3.95116349   0.0000000192412      0.0000002015918
NOTE: Optimal.
NOTE: Objective = 3.9511634887.
NOTE: Objective of the best feasible solution found = 3.9511579677.
NOTE: The best feasible solution found is returned.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 12 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 8 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 5 rows and 4 columns.
NOTE: Response BEST_FEASIBLE
```

(continues on next page)

(continued from previous page)

```
Problem Summary

                                Value
Label
Objective Sense          Minimization
Objective Function                  f
Objective Type              Nonlinear

Number of Variables                 8
Bounded Above                       0
Bounded Below                       0
Bounded Below and Above             8
Free                                0
Fixed                               0

Number of Constraints               5
Linear LE (<=)                      0
Linear EQ (=)                       0
Linear GE (>=)                      0
Linear Range                        0
Nonlinear LE (<=)                   0
Nonlinear EQ (=)                    0
Nonlinear GE (>=)                   4
Nonlinear Range                     1
Solution Summary

                            Value
Label
Solver                        NLP
Algorithm              Active Set
Objective Function              f
Solution Status     Best Feasible
Objective Value      3.9511579677

Optimality Error     0.0001050714
Infeasibility        7.7302351E-7

Iterations                     26
Presolve Time                0.00
Solution Time                0.02
   COL1         x
0   1.0  6.463315
1   2.0  2.234530
2   3.0  0.667455
3   4.0  0.595820
4   5.0  5.932980
5   6.0  5.527231
6   7.0  1.013787
7   8.0  0.400664
Out[2]: 3.951158
```

7.2.3 Nonlinear 2

Reference

http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_nlpsolver_examples02.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/nlpse02.html

Model

```
import sasoptpy as so
import sasoptpy.math as sm


def test(cas_conn):

    m = so.Model(name='nlpse02', session=cas_conn)
    N = m.add_parameter(name='N', init=1000)
    x = m.add_variables(so.exp_range(1, N), name='x', init=1)
    m.set_objective(
        so.quick_sum(-4*x[i]+3 for i in so.exp_range(1, N-1)) +
        so.quick_sum((x[i]**2 + x[N]**2)**2 for i in so.exp_range(1, N-1)),
        name='f', sense=so.MIN)

    m.add_statement('print x;', after_solve=True)
    m.solve(options={'with': 'nlp'}, verbose=True)
    if m.test_session() == 'CAS':
        print(m.response['Print3.PrintTable'])

    # Model 2
    so.reset_globals()
    m = so.Model(name='nlpse02_2', session=cas_conn)
    N = m.add_parameter(name='N', init=1000)
    x = m.add_variables(so.exp_range(1, N), name='x', lb=1, ub=2)
    m.set_objective(
        so.quick_sum(sm.cos(-0.5*x[i+1] - x[i]**2) for i in so.exp_range(
            1, N-1)), name='f2', sense=so.MIN)
    m.add_statement('print x;', after_solve=True)
    m.solve(verbose=True, options={'with': 'nlp', 'algorithm': 'activeset'})
    print(m.get_solution_summary())

    return m.get_objective_value()
```

Output

```
In [1]: from examples.nonlinear_2 import test

In [2]: test(cas_conn)
NOTE: Initialized model nlpse02.
NOTE: Added action set 'optimization'.
NOTE: Converting model nlpse02 to OPTMODEL.
num N init 1000;
var x {{1..N}} init 1;
```

(continues on next page)

(continued from previous page)

```
min f = sum {i_1 in 1..N-1}(- 4 * x[i_1] + 3) + sum {i_2 in 1..N-1}(((x[i_2]) ^ (2) +
↪(x[N]) ^ (2)) ^ (2));
solve with nlp;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
print x;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 1000 variables (1000 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0
↪nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using 2 threads for nonlinear evaluation.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
                           Objective                          Optimality
              Iter             Value     Infeasibility             Error
                 0     2997.00000000                 0     3996.00000000
                 1     2903.33927274                 0     3901.96888568
                 2     2720.89298022                 0     3716.59858581
                 3     2375.45256010                 0     3356.96682109
                 4     2050.78067864                 0     3007.33819156
                 5     1479.51953631                 0     2358.96117840
                 6      635.46851927                 0     1297.01852837
                 7       47.88027207                 0      263.90369702
                 8        0.56099667                 0       25.76387053
                 9        0.00010025                 0        0.31770898
                10   0.0000000000556                 0        0.00005787
                11                 0                 0   1.9350623913E-12
NOTE: Optimal.
NOTE: Objective = 0.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 12 rows
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 12 rows and
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 1000 rows and 6
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 0 rows and 4 columns.
       COL1             x
0       1.0  1.000000e+00
1       2.0  1.000000e+00
2       3.0  1.000000e+00
3       4.0  1.000000e+00
4       5.0  1.000000e+00
5       6.0  1.000000e+00
6       7.0  1.000000e+00
7       8.0  1.000000e+00
8       9.0  1.000000e+00
9      10.0  1.000000e+00
10     11.0  1.000000e+00
11     12.0  1.000000e+00
12     13.0  1.000000e+00
13     14.0  1.000000e+00
14     15.0  1.000000e+00
```

(continues on next page)

(continued from previous page)

```
15     16.0  1.000000e+00
16     17.0  1.000000e+00
17     18.0  1.000000e+00
18     19.0  1.000000e+00
19     20.0  1.000000e+00
20     21.0  1.000000e+00
21     22.0  1.000000e+00
22     23.0  1.000000e+00
23     24.0  1.000000e+00
24     25.0  1.000000e+00
25     26.0  1.000000e+00
26     27.0  1.000000e+00
27     28.0  1.000000e+00
28     29.0  1.000000e+00
29     30.0  1.000000e+00
..      ...           ...
970   971.0  1.000000e+00
971   972.0  1.000000e+00
972   973.0  1.000000e+00
973   974.0  1.000000e+00
974   975.0  1.000000e+00
975   976.0  1.000000e+00
976   977.0  1.000000e+00
977   978.0  1.000000e+00
978   979.0  1.000000e+00
979   980.0  1.000000e+00
980   981.0  1.000000e+00
981   982.0  1.000000e+00
982   983.0  1.000000e+00
983   984.0  1.000000e+00
984   985.0  1.000000e+00
985   986.0  1.000000e+00
986   987.0  1.000000e+00
987   988.0  1.000000e+00
988   989.0  1.000000e+00
989   990.0  1.000000e+00
990   991.0  1.000000e+00
991   992.0  1.000000e+00
992   993.0  1.000000e+00
993   994.0  1.000000e+00
994   995.0  1.000000e+00
995   996.0  1.000000e+00
996   997.0  1.000000e+00
997   998.0  1.000000e+00
998   999.0  1.000000e+00
999  1000.0  9.684997e-16

[1000 rows x 2 columns]
NOTE: Initialized model nlpse02_2.
NOTE: Added action set 'optimization'.
NOTE: Converting model nlpse02_2 to OPTMODEL.
num N init 1000;
var x {{1..N}} >= 1 <= 2;
min f2 = sum {i_1 in 1..N-1}(cos(- 0.5 * x[i_1 + 1] - (x[i_1]) ^ (2)));
solve with nlp / algorithm=activeset;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
```

(continues on next page)

(continued from previous page)

```
print x;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 80 threads.
NOTE: The problem has 1000 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0␣
↪nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: Using 3 threads for nonlinear evaluation.
NOTE: The NLP solver is called.
NOTE: The Active Set algorithm is used.
NOTE: Initial point was changed to be feasible to bounds.
                           Objective                          Optimality
              Iter             Value     Infeasibility             Error
                 0       70.66646447                 0        1.24686873
                 1       70.66646439                 0        1.24686873
                 2     -996.26893548                 0        0.23815533
                 3     -998.99328004                 0        0.10718277
                 4     -998.99999439                 0        0.00379400
                 5     -999.00000000                 0        0.00000393
                 6     -999.00000000                 0   1.7018480129E-12
NOTE: Optimal.
NOTE: Objective = -999.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 12 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 12 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 1000 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 0 rows and 4 columns.
Solution Summary

                           Value
Label
Solver                       NLP
Algorithm             Active Set
Objective Function            f2
Solution Status          Optimal
Objective Value             -999

Optimality Error    1.701848E-12
Infeasibility                  0

Iterations                     6
Presolve Time               0.00
Solution Time               0.13
Out[2]: -999.0
```

7.3 SAS (saspy) Examples

7.3.1 Decentralization (saspy)

Reference

http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex10_toc.htm&docsetVersion=14.3&locale=en

http://support.sas.com/documentation/onlinedoc/or/ex_code/143/mpex10.html

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(
            ['i', 'j'])

    max_num_depts = 3

    benefit = {}
    for city in CITIES:
        for dept in DEPTS:
```

(continues on next page)

(continued from previous page)

```
        try:
            benefit[dept, city] = benefit_data.loc[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, name='product')

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='pd1')

product_def2 = m.add_constraints((product[i, j, k, l] <= assign[i, j]
                                  for (i, j, k, l) in IJKL),
                                 name='pd2')

product_def3 = m.add_constraints((product[i, j, k, l] <= assign[k, l]
                                  for (i, j, k, l) in IJKL),
                                 name='pd3')

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

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

(continues on next page)

(continued from previous page)

```
    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='pd4')

    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='pd5')

    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.decentralization import test
>>> sas_session = saspy.SASsession(cfgname='winlocal')
>>> test(sas_session)

SAS Connection established. Subprocess id is 18384

NOTE: Initialized model decentralization.
NOTE: Converting model decentralization to OPTMODEL.
NOTE: Submitting OPTMODEL codes to SAS server.
NOTE: Writing HTML5(SASPY_INTERNAL) Body file: _TOMODS1
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 105 variables (0 free, 0 fixed).
NOTE: The problem has 105 binary and 0 integer variables.
NOTE: The problem has 278 linear constraints (183 LE, 5 EQ, 90 GE, 0 range).
NOTE: The problem has 660 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
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    135.0000000  111.04%       0
             0        1      3    -14.9000000     67.5000000  122.07%       0
             0        1      3    -14.9000000     55.0000000  127.09%       0
```

(continues on next page)

(continued from previous page)

```
             0        1      4      8.1000000     48.0000000   83.12%       0
             0        1      4      8.1000000     44.8375000   81.93%       0
             0        1      4      8.1000000     42.0000000   80.71%       0
             0        1      4      8.1000000     39.0666667   79.27%       0
             0        1      4      8.1000000     34.7500000   76.69%       0
             0        1      4      8.1000000     33.9000000   76.11%       0
             0        1      4      8.1000000     29.6800000   72.71%       0
             0        1      4      8.1000000     28.5000000   71.58%       0
             0        1      4      8.1000000     28.5000000   71.58%       0
             0        1      4      8.1000000     28.5000000   71.58%       0
             0        1      4      8.1000000     28.5000000   71.58%       0
NOTE: The MILP solver added 31 cuts with 168 cut coefficients at the root.
             2        0      5     14.9000000     14.9000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The data set WORK.PROB_SUMMARY has 20 observations and 3 variables.
NOTE: The data set WORK.SOL_SUMMARY has 18 observations and 3 variables.
NOTE: The data set WORK.PRIMAL_OUT has 105 observations and 6 variables.
NOTE: The data set WORK.DUAL_OUT has 278 observations and 4 variables.
NOTE: PROCEDURE OPTMODEL used (Total process time):
      real time           0.34 seconds
      cpu time            0.29 seconds

                              Value
Label
Objective Sense          Maximization
Objective Function         netBenefit
Objective Type                 Linear

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

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

Constraint Coefficients           660
NOTE: Converting model decentralization to OPTMODEL.
NOTE: Submitting OPTMODEL codes to SAS server.
NOTE: Writing HTML5(SASPY_INTERNAL) Body file: _TOMODS1
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 105 variables (0 free, 0 fixed).
NOTE: The problem has 105 binary and 0 integer variables.
NOTE: The problem has 68 linear constraints (3 LE, 65 EQ, 0 GE, 0 range).
NOTE: The problem has 270 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
```

(continues on next page)

(continued from previous page)

```
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    -28.1000000    135.0000000  120.81%       0
             0        1      3    -28.1000000     30.0000000  193.67%       0
             0        1      4    -16.3000000     30.0000000  154.33%       0
             0        1      5     14.9000000     14.9000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The data set WORK.PROB_SUMMARY has 20 observations and 3 variables.
NOTE: The data set WORK.SOL_SUMMARY has 18 observations and 3 variables.
NOTE: The data set WORK.PRIMAL_OUT has 105 observations and 6 variables.
NOTE: The data set WORK.DUAL_OUT has 68 observations and 4 variables.
NOTE: PROCEDURE OPTMODEL used (Total process time):
      real time           0.19 seconds
      cpu time            0.14 seconds

                                Value
Label
Objective Sense          Maximization
Objective Function         netBenefit
Objective Type                 Linear

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

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

7.4 RESTful API Examples

7.4.1 Knapsack

Model

```
import requests
import json

import sasoptpy as so
from sasoptpy.api import api
so.reset_globals()

def test(cashost, port):

    # Start server
    api.start(thread=True, host='127.0.0.1', port=5000)

    host = 'http://127.0.0.1:5000'

    # Get server and version info
    res = requests.get(host)

    # Create new workspace
    res = requests.post(host + '/workspaces',
                        data={'name': 'myworkspace',
                              'password': 12345})

    # If workspace exists, renew the token
    res = requests.post(host + '/workspaces/myworkspace',
                        data={'password': 12345})

    # Save the token
    token = 'Bearer ' + res.json()['token']
    headers = {'Authorization': token}

    # Clean workspace
    res = requests.post(host, data={'action': 'clean'}, headers=headers)

    # Create a new CAS session
    res = requests.post(host + '/sessions', headers=headers,
                        data={'name': 'mycas', 'host': cashost, 'port': port})

    # Create a new model
    res = requests.post(host + '/models', headers=headers,
                        data={'name': 'knapsack', 'session': 'mycas'})

    # Create variables
    res = requests.post(host + '/models/knapsack/variable_groups', headers=headers,
                        json={'name': 'pick', 'index': [["pen","watch","cup"]],
↪'vartype': 'integer'})

    # Set objective function
    res = requests.post(host + '/models/knapsack/objectives', headers=headers,
                        json={'expression': "5*pick['pen']+20*pick['watch']+2*pick[
↪'cup']", 'sense': 'maximize',
```

(continues on next page)

(continued from previous page)

```
                                 'name': 'total_value'})

    # Capacity constraint
    res = requests.post(host + '/models/knapsack/constraints', headers=headers,
                        json={'expression': "1*pick['pen']+3*pick['watch']+10*pick[
↪'cup']<=22", 'name': 'total_weight'})

    # Individual limits for items
    res = requests.post(host + '/models/knapsack/constraint_groups', headers=headers,
                        json={
                            'expression': 'pick[i]<=5', 'index': "for i in ['pen',
↪'watch','cup']",
                            'name': 'bounds'})

    # Get optmodel code of the model
    res = requests.get(host+'/models/knapsack', headers=headers,
                       params={'format': 'optmodel'})
    print(res.json()['optmodel'])

    # Solve the model
    res = requests.post(host + '/models/knapsack/solutions', headers=headers,
                        data={'stream': False})
    sols = res.json()['solutions']
    for i in sols:
        print(i, sols[i])
```

Output

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

In [2]: test(hostname, port)
Starting WSGI Server...
NOTE: Initialized model knapsack.
proc optmodel;
var pick {{'pen','watch','cup'}} integer >= 0;
maximize total_value = 5 * pick['pen'] + 20 * pick['watch'] + 2 * pick['cup'];
con total_weight : pick['pen'] + 3 * pick['watch'] + 10 * pick['cup'] <= 22;
con bounds_pen : pick['pen'] <= 5;
con bounds_watch : pick['watch'] <= 5;
con bounds_cup : pick['cup'] <= 5;

solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
quit;

NOTE: Added action set 'optimization'.
NOTE: Converting model knapsack to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 32 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 0 binary and 3 integer variables.
NOTE: The problem has 4 linear constraints (4 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
```

(continues on next page)

(continued from previous page)

```
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 3 constraints.
NOTE: The MILP presolver removed 3 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 3 variables, 1 constraints, and 3 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    125.0000000    125.9990000    0.79%       0
                0        0      3    125.0000000    125.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 125.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSERHDFS(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSERHDFS(casuser)' has 20 rows and␣
↪4 columns.
NOTE: The CAS table 'primal' in caslib 'CASUSERHDFS(casuser)' has 3 rows and 6␣
↪columns.
NOTE: The CAS table 'dual' in caslib 'CASUSERHDFS(casuser)' has 4 rows and 4 columns.
pick[pen] 5.0
pick[watch] 5.0
pick[cup] -0.0
```

CHAPTER

EIGHT

API REFERENCE

8.1 Model

8.1.1 Constructor

Model(name[, session])	Creates an optimization model

sasoptpy.Model

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

Bases: object

Creates an optimization model

Parameters

name [string] Name of the model

session [swat.cas.connection.CAS or saspy.SASsession, 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
```

8.1.2 Components

Model.set_session(session)	Sets the CAS session for model
Model.add_constraint(c[, name])	Adds a single constraint to the model
Model.add_constraints(argv[, cg, name])	Adds a set of constraints to the model

Continued on next page

Table 2 – continued from previous page

Model.add_variable([var, vartype, name, lb, ...])	Adds a new variable to the model
Model.add_variables(*argv[, vg, name, ...])	Adds a group of variables to the model
Model.add_implicit_variable([argv, name])	Adds an implicit variable to the model
Model.add_set(name[, init, value, settype])	Adds a set to the model
Model.add_parameter(*argv[, name, init, ...])	Adds a parameter to the model
Model.add_statement(statement[, after_solve])	Adds a PROC OPTMODEL statement to the model
Model.set_objective(expression[, sense, ...])	Sets the objective function for the model
Model.set_coef(var, con, value)	Updates the coefficient of a variable inside constraints
Model.drop_constraint(constraint)	Drops a constraint from the model
Model.drop_constraints(constraints)	Drops a constraint group from the model
Model.drop_variable(variable)	Drops a variable from the model
Model.drop_variables(variables)	Drops a variable group from the model
Model.get_constraint(name)	Returns the reference to a constraint in the model
Model.get_constraints()	Returns a list of constraints in the model
Model.get_variable(name)	Returns the reference to a variable in the model
Model.get_variables()	Returns a list of variables
Model.get_objective()	Returns the objective function as an Expression object
Model.get_variable_coef(var)	Returns the objective value coefficient of a variable
Model.read_data(table, key_set[, key_cols, ...])	Reads a CASTable into PROC OPTMODEL and adds it to the model
Model.read_table(table[, key, key_type, ...])	Reads a CAS Table or pandas DataFrame into the model
Model.include(*argv)	Adds existing variables and constraints to a model

sasoptpy.Model.set_session

`Model.set_session(session)`

Sets the CAS session for model

Parameters

session [swat.cas.connection.CAS or saspy.SASsession] CAS or SAS Session object

Notes

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

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

c [Constraint] Reference to the constraint

See also:

Constraint, *Model.include()*

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 object] List of constraints as a Generator-type object

cg [ConstraintGroup, 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

cg [ConstraintGroup] Reference to the ConstraintGroup

See also:

ConstraintGroup, *Model.include()*

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]
```

(continues on next page)

(continued from previous page)

```
  [(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]
  [(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=-inf*, *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, 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

var [Variable] Variable that is added to the model

See also:

Variable, *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*, *abstract=None*)

Adds a group of variables to the model

Parameters

argv [list, dict, `pandas.Index`] Loop index for variable group

vg [VariableGroup, 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`, *`Model.include()`*

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.add_implicit_variable

`Model.add_implicit_variable`(*argv=None*, *name=None*)

Adds an implicit variable to the model

Parameters

argv [Generator-type object] Generator object where each item is an entry

name [string, optional] Name of the implicit variable

Notes

- Based on whether generated by a regular expression or an abstract one, implicit variables may appear in generated OPTMODEL codes.

Examples

```
>>> x = m.add_variables(range(5), name='x')
>>> y = m.add_implicit_variable((
>>>     x[i] + 2 * x[i+1] for i in range(4)), name='y')
>>> print(y[2])
x[2] + 2 * x[3]
```

```
>>> I = m.add_set(name='I')
>>> z = m.add_implicit_variable((x[i] * 2 + 2 for i in I), name='z')
>>> print(z._defn())
impvar z {i_1 in I} = 2 * x[i_1] + 2;
```

sasoptpy.Model.add_set

Model.**add_set**(*name, init=None, value=None, settype=['num']*)

Adds a set to the model

Parameters

name [string, optional] Name of the set

init [*Set*, optional] Initial value of the set

value [list, float, optional] Exact value of the set

settype [list, optional] Types of the set, a list consists of 'num' and 'str' values

Examples

```
>>> I = m.add_set(name='I')
>>> print(I._defn())
set I;
```

```
>>> J = m.add_set(name='J', settype=['str'])
>>> print(J._defn())
set <str> J;
```

```
>>> N = m.add_parameter(name='N', init=4)
>>> K = m.add_set(name='K', init=so.exp_range(1, N))
>>> print(K._defn())
set K = 1..N;
```

sasoptpy.Model.add_parameter

Model.**add_parameter**(**argv*, *name=None*, *init=None*, *value=None*, *p_type=None*)
Adds a parameter to the model

Parameters

argv [*Set*, optional] Key set of the parameter

name [string, optional] Name of the parameter

init [float or expression, optional] Initial value of the parameter

p_type [string, optional] Type of the parameter, 'num' for floats or 'str' for strings

Examples

```
>>> I = m.add_set(name='I')
>>> a = m.add_parameter(I, name='a', init=5)
>>> print(a._defn())
num a {I} init 5 ;
```

sasoptpy.Model.add_statement

Model.**add_statement**(*statement*, *after_solve=False*)
Adds a PROC OPTMODEL statement to the model

Parameters

statement [Expression or string] Statement object

after_solve [boolean, optional] Option for putting the statement after 'solve' declaration

Notes

- If the statement string includes 'print', then it is automatically placed after solve.
- The first parameter, *statement* could be a Statement object when internally used.

Examples

```
>>> I = m.add_set(name='I')
>>> x = m.add_variables(I, name='x', vartype=so.INT)
>>> a = m.add_parameter(I, name='a')
>>> c = m.add_constraints((x[i] <= 2 * a[i] for i in I), name='c')
>>> m.add_statement('print x;', after_solve=True)
>>> print(m.to_optmodel())
proc optmodel;
min m_obj = 0;
set I;
var x {I} integer >= 0;
num a {I};
con c {i_1 in I} : x[i_1] - 2.0 * a[i_1] <= 0;
solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
```

(continues on next page)

(continued from previous page)

```
print _con_.name _con_.body _con_.dual;
print x;
quit;
```

sasoptpy.Model.set_objective

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

Sets the objective function for the model

Parameters

expression [Expression] The objective function as an Expression

sense [string, optional] Objective value direction, 'MIN' or 'MAX'

name [string, optional] Name of the objective value

multiobj [boolean, optional] Option for keeping the objective function when working with multiple objectives

Returns

objective [Expression] Objective function as an `Expression` object

Notes

- Default objective sense is minimization (MIN)
- This method replaces the existing objective of the model. When working with multiple objectives, use the *multiobj* parameter and use 'obj' option in `Model.solve()` and `Model.to_optmodel()` methods.

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')
```

```
>>> m.set_objective(2 * x + y, sense=so.MIN, name='f1', multiobj=True)
>>> m.set_objective( (x - y) ** 2, sense=so.MIN, name='f2', multiobj=True)
>>> print(m.to_optmodel(options={'with': 'lso', 'obj': (f1, f2)}))
proc optmodel;
var x;
var y;
min f1 = 2 * x + y;
min f2 = (x - y) ^ (2);
solve with lso obj (f1 f2);
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
quit;
```

sasoptpy.Model.set_coef

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

Updates the coefficient of a variable inside constraints

Parameters

var [Variable] Variable whose coefficient will be updated

con [Constraint] 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.drop_constraint

`Model.drop_constraint(constraint)`

Drops a constraint from the model

Parameters

constraint [Constraint] 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 * 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] 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')]
>>> 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] 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] 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 [Constraint] Reference to the constraint

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

constraints [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_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 [Variable] Reference to the variable

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_variables

Model.**get_variables**()

Returns a list of variables

Returns

variables [list] List of variables in the model

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.get_objective

Model.**get_objective**()

Returns the objective function as an *Expression* object

Returns

objective [Expression] 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_variable_coef

`Model.get_variable_coef(var)`

Returns the objective value coefficient of a variable

Parameters

var [Variable or string] Variable whose objective value is requested or its name

Returns

coef [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.read_data

`Model.read_data(table, key_set, key_cols=None, option=”, params=None)`

Reads a CASTable into PROC OPTMODEL and adds it to the model

Parameters

table [`swat.cas.table.CASTable`] The CAS table to be read to sets and parameters

key_set [`Set`] Set object to be read as the key (index)

key_cols [list or string, optional] Column names of the key columns

option [string, optional] Additional options for read data command

params [list, optional] A list of dictionaries where each dictionary represent parameters

See also:

`sasoptpy.utils.read_data()`

Notes

- This function is intended to be used internally.
- It imitates the `read data` statement of PROC OPTMODEL.
- This function is still under development and subject to change.
- *key_cols* parameters should be a list. When passing a single item, string type can be used instead.
- Values inside each dictionary in `params` list should be as follows:
  - **param** : `Parameter`

    Paramter object, whose index is the same as table key

- **column** : string, optional

  Column name to be read

- **index** : list, optional

  List of sets if the parameter has to be read in a loop

Examples

```
>>> table = session.upload_frame(df, casout='df')
>>> item = m.add_set(name='set_item')
>>> value = m.add_parameter(item, name='value')
>>> m.read_data(table, key_set=item, key_cols=['items'],params=[{'param': value,
↪'column': 'value'}])
>>> print(m.to_optmodel())
proc optmodel;
min m_obj = 0;
set set_item;
num value {set_item};
read data df into set_item=[items] value;
solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
quit;
```

sasoptpy.Model.read_table

`Model.`**`read_table`**(*table, key=['_N_'], key_type=['num'], key_name=None, columns=None, col_types=None, col_names=None, upload=False, casout=None*)

Reads a CAS Table or pandas DataFrame into the model

Parameters

table [`swat.cas.table.CASTable`, `pandas.DataFrame` or string] Pointer to CAS Table (server data, CASTable), DataFrame (local data) or the name of the table at execution (server data, string)

key [list, optional] List of key columns (for CASTable) or index columns (for DataFrame)

key_type [list or string, optional] A list of column types consists of 'num' or 'str' values

key_name [string, optional] Name of the key set

columns [list, optional] List of columns to read into parameters

col_types [dict, optional] Dictionary of column types

col_names [dict, optional] Dictionary of column names

upload [boolean, optional] Option for uploading a local data to CAS server first

casout [string or dict, optional] Casout options if data is uploaded

Returns

objs [tuple] A tuple where first element is the key (index) and second element is a list of requested columns

See also:

`Model.read_data()`, *`Model.add_parameter()`*, *`Model.add_set()`*

Notes

- This method can take either a `swat.cas.table.CASTable`, a `pandas.DataFrame` or name of the data set as a string as the first argument.
- If the model is running in saspy or MPS mode, then the data is read to client from the CAS server.
- If the model is running in OPTMODEL mode, then this method generates the corresponding optmodel code.
- When table is a CASTable object, since the actual data is stored on the CAS server, some of the functionalities may be limited.
- For the local data, `upload` argument can be passed for performance improvement.
- See `swat.CAS.upload_frame()` and `table.loadtable` CAS action for `casout` options.

Examples

```
>>> info = pd.DataFrame([
        ['clock', 6, 15, 1],
        ['pc', 3, 14, 5],
        ['headphone', 2, 9, 3],
        ['mug', 2, 4, 1],
        ['book', 5, 1, 3],
        ['pen', 1, 1, 4]
        ], columns=['item', 'weight', 'value', 'limit'])
>>> ITEMS, [weight, value, limit] = m.read_table(
        info, key=['item'], columns=['weight', 'value', 'limit'],
        key_stype='str', upload=False)
```

sasoptpy.Model.include

`Model.include(*argv)`

Adds existing variables and constraints to a model

Parameters

argv [*Model*, *Variable*, *Constraint*, *VariableGroup*, *ConstraintGroup*, *Set*, *Parameter*, Statement, *ImplicitVar*] Objects to be included in the model

Notes

- 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 all of its elements)

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

8.1.3 Solver calls

`Model.solve`([options, submit, name, frame, . . .])	Solves the model by calling CAS or SAS optimization solvers
`Model.solve_on_cas`(session, options, submit, . . .)	Solves the optimization problem on CAS Servers
`Model.solve_on_mva`(session, options, submit, . . .)	Solves the optimization problem on SAS Clients
`Model.get_solution`([vtype, solution, pivot])	Returns the solution details associated with the primal or dual solution
`Model.get_variable_value`([var, name])	Returns the value of a variable.
`Model.get_objective_value`()	Returns the optimal objective value, if it exists
`Model.get_solution_summary`()	Returns the solution summary table to the user
`Model.get_problem_summary`()	Returns the problem summary table to the user
`Model.print_solution`()	Prints the current values of the variables
`Model.upload_user_blocks`()	Uploads user-defined decomposition blocks to the CAS server

sasoptpy.Model.solve

`Model.`**`solve`**(*options=None*, *submit=True*, *name=None*, *frame=False*, *drop=False*, *replace=True*, *primalin=False*, *milp=None*, *lp=None*, *verbose=False*)

Solves the model by calling CAS or SAS optimization solvers

Parameters

options [dict, optional] A dictionary solver options

submit [boolean, optional] Switch for calling the solver instantly

name [string, optional] Name of the table name

frame [boolean, optional] Switch for uploading problem as a MPS DataFrame format

drop [boolean, optional] Switch for dropping the MPS table after solve (only CAS)

replace [boolean, optional] Switch for replacing an existing MPS table (only CAS and MPS)

primalin [boolean, optional] Switch for using initial values (only MILP)

verbose [boolean, optional (experimental)] Switch for printing generated OPTMODEL code

Returns

solution [`pandas.DataFrame`] Solution of the optimization model

See also:

`Model.solve_on_cas()`, *`Model.solve_on_mva()`*

Notes

- This method is essentially a wrapper for two other methods.
- Some of the options listed under `options` argument may not be passed based on which CAS Action is being used.
- The `option` argument should be a dictionary, where keys are option names. For example, `m.solve(options={'maxtime': 600})` limits the solution time to 600 seconds.
- See *Solver Options* for a list of solver 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(options={'maxtime': 600})
```

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

sasoptpy.Model.solve_on_cas

`Model.`**`solve_on_cas`**(*session, options, submit, name, frame, drop, replace, primalin, verbose*)

Solves the optimization problem on CAS Servers

See also:

`Model.solve()`

Notes

- This function should not be called directly. Instead, use *`Model.solve()`*.

sasoptpy.Model.solve_on_mva

`Model.solve_on_mva`(*session*, *options*, *submit*, *name*, *frame*, *drop*, *replace*, *primalin*, *verbose*)

Solves the optimization problem on SAS Clients

See also:

Model.solve()

Notes

- This function should not be called directly. Instead, use *Model.solve()*.

sasoptpy.Model.get_solution

`Model.get_solution`(*vtype='Primal'*, *solution=None*, *pivot=False*)

Returns the solution details associated with the primal or dual solution

Parameters

vtype [string, optional] 'Primal' or 'Dual'

solution [integer, optional] Solution number to be returned (for MILP)

pivot [boolean, optional] Switch for returning multiple solutions in columns as a pivot table

Returns

solution [`pandas.DataFrame`] Primal or dual solution table returned from the CAS Action

Notes

- If *Model.solve()* method is used with `frame=True` option, MILP solver returns multiple solutions. You can obtain different results using `solution` parameter.

Examples

```
>>> m.solve()
>>> print(m.get_solution('Primal'))
              var   lb            ub  value  solution
0        x[clock]  0.0  1.797693e+308   0.0       1.0
1           x[pc]  0.0  1.797693e+308   5.0       1.0
2    x[headphone]  0.0  1.797693e+308   2.0       1.0
3          x[mug]  0.0  1.797693e+308   0.0       1.0
4         x[book]  0.0  1.797693e+308   0.0       1.0
5          x[pen]  0.0  1.797693e+308   1.0       1.0
6        x[clock]  0.0  1.797693e+308   0.0       2.0
7           x[pc]  0.0  1.797693e+308   5.0       2.0
8    x[headphone]  0.0  1.797693e+308   2.0       2.0
9          x[mug]  0.0  1.797693e+308   0.0       2.0
10        x[book]  0.0  1.797693e+308   0.0       2.0
11         x[pen]  0.0  1.797693e+308   0.0       2.0
12       x[clock]  0.0  1.797693e+308   1.0       3.0
13          x[pc]  0.0  1.797693e+308   4.0       3.0
...
```

```
>>> print(m.get_solution('Primal', solution=2))
              var   lb            ub  value  solution
6        x[clock]  0.0  1.797693e+308    0.0       2.0
7           x[pc]  0.0  1.797693e+308    5.0       2.0
8    x[headphone]  0.0  1.797693e+308    2.0       2.0
9          x[mug]  0.0  1.797693e+308    0.0       2.0
10        x[book]  0.0  1.797693e+308    0.0       2.0
11         x[pen]  0.0  1.797693e+308    0.0       2.0
```

```
>>> print(m.get_solution(pivot=True))
solution      1.0  2.0  3.0  4.0  5.0
var
x[book]       0.0  0.0  0.0  1.0  0.0
x[clock]      0.0  0.0  1.0  1.0  0.0
x[headphone]  2.0  2.0  1.0  1.0  0.0
x[mug]        0.0  0.0  0.0  1.0  0.0
x[pc]         5.0  5.0  4.0  1.0  0.0
x[pen]        1.0  0.0  0.0  1.0  0.0
```

```
>>> print(m.get_solution('Dual'))
                       con  value  solution
0              weight_con   20.0       1.0
1        limit_con[clock]    0.0       1.0
2           limit_con[pc]    5.0       1.0
3    limit_con[headphone]    2.0       1.0
4          limit_con[mug]    0.0       1.0
5         limit_con[book]    0.0       1.0
6          limit_con[pen]    1.0       1.0
7              weight_con   19.0       2.0
8        limit_con[clock]    0.0       2.0
9           limit_con[pc]    5.0       2.0
10   limit_con[headphone]    2.0       2.0
11         limit_con[mug]    0.0       2.0
12        limit_con[book]    0.0       2.0
13         limit_con[pen]    0.0       2.0
...
```

```
>>> print(m.get_solution('dual', pivot=True))
solution               1.0   2.0   3.0   4.0  5.0
con
limit_con[book]        0.0   0.0   0.0   1.0  0.0
limit_con[clock]       0.0   0.0   1.0   1.0  0.0
limit_con[headphone]   2.0   2.0   1.0   1.0  0.0
limit_con[mug]         0.0   0.0   0.0   1.0  0.0
limit_con[pc]          5.0   5.0   4.0   1.0  0.0
limit_con[pen]         1.0   0.0   0.0   1.0  0.0
weight_con            20.0  19.0  20.0  19.0  0.0
```

sasoptpy.Model.get_variable_value

`Model.`**`get_variable_value`**(*var=None*, *name=None*)

Returns the value of a variable.

Parameters

var [:Variable, optional] Variable object

name [string, optional] Name of the variable

Notes

- It is possible to get a variable's value using *Variable.get_value()* method, if the variable is not abstract.
- This method is a wrapper around *Variable.get_value()* and an overlook function for model components

sasoptpy.Model.get_objective_value

Model.**get_objective_value**()

Returns the optimal objective value, if it exists

Returns

objective_value [float] Objective value at current solution

Notes

- This method should be used for getting the objective value after solve. Using `m.get_objective().get_value()` actually evaluates the expression using optimal variable values. This may not be available for nonlinear expressions.

Examples

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

sasoptpy.Model.get_solution_summary

Model.**get_solution_summary**()

Returns the solution summary table to the user

Returns

ss [`swat.dataframe.SASDataFrame`] 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
```

(continues on next page)

(continued from previous page)

```
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_problem_summary

`Model.get_problem_summary()`

Returns the problem summary table to the user

Returns

ps [`swat.dataframe.SASDataFrame`] 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
```

(continues on next page)

(continued from previous page)

```
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',
'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.print_solution

Model.**print_solution**()

Prints the current values of the variables

See also:

Model.get_solution()

Notes

- This function may not work for abstract variables and nonlinear models.

Examples

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

sasoptpy.Model.upload_user_blocks

Model.**upload_user_blocks**()

Uploads user-defined decomposition blocks to the CAS server

Returns

name [string] CAS table name of the user-defined decomposition blocks

Examples

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

8.1.4 Export

Model.to_frame([constant])	Converts the Python model into a DataFrame object in MPS format
Model.to_optmodel([header, expand, ordered, ...])	Generates the equivalent PROC OPTMODEL code for the model.

sasoptpy.Model.to_frame

`Model.to_frame`(*constant=False*)

Converts the Python model into a DataFrame object in MPS format

Parameters

constant [boolean, optional] Switching for using objConstant argument for solveMilp, solveLp. Adds the constant as an auxiliary variable if value is True.

Returns

mpsdata [`pandas.DataFrame`] Problem representation 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.to_optmodel

`Model.to_optmodel(header=True, expand=False, ordered=False, ods=False, solve=True, options={}, primalin=False)`

Generates the equivalent PROC OPTMODEL code for the model.

Parameters

header [boolean, optional] Option to include PROC headers

expand [boolean, optional] Option to include 'expand' command to OPTMODEL code

ordered [boolean, optional] Option to generate OPTMODEL code in a specific order (True) or in creation order (False)

ods [boolean, optional] Option for converting printed tables into ODS tables

solve [boolean, optional] Option to append solve command at the end

options [dict, optional] Solver options for the OPTMODEL solve command

Returns

s [string] PROC OPTMODEL representation of the model

Notes

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

Examples

```
>>> print(m.to_optmodel())
proc optmodel;
var get {{'clock','mug','headphone','book','pen'}} integer >= 0;
get['clock'] = 3.0;
get['mug'] = 4.0;
get['headphone'] = 2.0;
get['book'] = -0.0;
get['pen'] = 5.0;
con limit_con_clock : get['clock'] <= 3;
con limit_con_mug : get['mug'] <= 5;
con limit_con_headphone : get['headphone'] <= 2;
con limit_con_book : get['book'] <= 10;
con limit_con_pen : get['pen'] <= 15;
con weight_con : 4 * get['clock'] + 6 * get['mug'] + 7 * get['headphone'] + 12 *
↪get['book'] + get['pen'] <= 55;
max total_value = 8 * get['clock'] + 10 * get['mug'] + 15 * get['headphone'] + 20
↪* get['book'] + get['pen'];
solve;
print _var_.name _var_.lb _var_.ub _var_ _var_.rc;
print _con_.name _con_.body _con_.dual;
quit;
```

8.1.5 Internal functions

`Model.upload_model`([name, replace, constant])	Converts internal model to MPS table and upload to CAS session
`Model.test_session`()	Tests if the model session is defined and still active
`Model._is_linear`()	Checks if the model can be written as a linear model (in MPS format)

sasoptpy.Model.upload_model

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

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

frame [`swat.cas.table.CASTable`] 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.test_session

`Model.test_session`()

Tests if the model session is defined and still active

Returns

session [string] 'CAS' for CAS sessions, 'SAS' for SAS sessions, None otherwise

sasoptpy.Model._is_linear

`Model._is_linear`()

Checks if the model can be written as a linear model (in MPS format)

Returns

is_linear [boolean] True if model does not have any nonlinear components or abstract operations, False otherwise

8.2 Expression

8.2.1 Constructor

Expression([exp, name, temp])	Creates a mathematical expression to represent model components

sasoptpy.Expression

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

Bases: object

Creates a mathematical expression to represent model components

Parameters

exp [Expression, 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)
```

```
>>> import sasoptpy.math as sm
>>> f = sm.sin(x) + sm.min(y[1],1) ** 2
>>> print(type(f))
<class 'sasoptpy.components.Expression'>
>>> print(f)
sin(x) + (min(y[1] , 1)) ** (2)
```

8.2.2 Methods

`Expression.add`(other[, sign])	Combines two expressions and produces a new one
`Expression.copy`([name])	Returns a copy of the `Expression` object
`Expression.get_name`()	Returns the name of the expression
`Expression.get_value`()	Calculates and returns the value of the linear expression
`Expression.mult`(other)	Multiplies the `Expression` with a scalar value
`Expression.set_name`([name])	Sets the name of the expression
`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] Second expression or constant value to be added

sign [int, optional] Sign of the addition, 1 or -1

Returns

r [Expression] Reference to the outcome of the operation

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

r [Expression] Copy of the object

Examples

```
>>> x = so.Variable(name='x')
>>> y = so.VariableGroup(1, name='y')
>>> 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_name

`Expression.get_name()`

Returns the name of the expression

Returns

name [string] Name of the expression

Examples

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

sasoptpy.Expression.get_value

`Expression.get_value()`

Calculates and returns the value of the linear expression

Returns

v [float] Value of the expression

Examples

```
>>> sales = so.Variable(name='sales', init=10)
>>> material = so.Variable(name='material', init=3)
>>> profit = so.Expression(5 * sales - 3 * material)
>>> print(profit.get_value())
41
```

sasoptpy.Expression.mult

`Expression.mult(other)`

Multiplies the `Expression` with a scalar value

Parameters

other [Expression or int] Second expression to be multiplied

Returns

r [Expression] 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=None)`

Sets the name of the expression

Parameters

name [string] Name of the expression

Returns

name [string] Name of the expression after resolving conflicts

Examples

```
>>> x = so.Variable(name='x')
>>> e = x**2 + 2*x + 1
>>> e.set_name('expansion')
```

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

name [string] Name of the expression in the namespace

8.2.3 Private Methods

`Expression._expr()`	Generates the OPTMODEL compatible string representation of the object.
`Expression._is_linear()`	Checks if the expression is composed of linear components
`Expression._relational(other, direction_)`	Creates a logical relation between `Expression` objects
`Expression.__repr__()`	Returns a string representation of the object.
`Expression.__str__()`	Generates a representation string that is Python compatible

sasoptpy.Expression._expr

`Expression._expr()`

Generates the OPTMODEL compatible string representation of the object.

Examples

```
>>> x = so.Variable(name='x')
>>> y = so.Variable(name='y')
>>> f = x + y ** 2
>>> print(f)
x + (y) ** (2)
>>> print(f._expr())
x + (y) ^ (2)
```

sasoptpy.Expression._is_linear

Expression.**_is_linear**()
Checks if the expression is composed of linear components

Returns

is_linear [boolean] True if the expression is linear, False otherwise

Examples

```
>>> x = so.Variable()
>>> e = x*x
>>> print(e.is_linear())
False
```

```
>>> f = x*x + x*x - 2*x*x + 5
>>> print(f.is_linear())
True
```

sasoptpy.Expression._relational

Expression.**_relational**(*other*, *direction_*)
Creates a logical relation between *Expression* objects

Parameters

other [Expression] Expression on the other side of the relation wrt self

direction_ [string] Direction of the logical relation, either 'E', 'L', or 'G'

Returns

generated_constraint [Constraint] Constraint generated as a result of linear relation

sasoptpy.Expression.__repr__

Expression.**__repr__**()
Returns a string representation of the object.

Examples

```
>>> x = so.Variable(name='x')
>>> y = so.Variable(name='y')
>>> f = x + y ** 2
>>> print(repr(f))
sasoptpy.Expression(exp = x + (y) ** (2), name=None)
```

sasoptpy.Expression.__str__

Expression.**__str__**()
Generates a representation string that is Python compatible

Examples

```
>>> f = x + y ** 2
>>> print(str(f))
x + (y) ** (2)
```

8.3 Variable

8.3.1 Constructor

Variable(name[, vartype, lb, ub, init, . . .])	Creates an optimization variable to be used inside models

sasoptpy.Variable

class sasoptpy.**Variable**(*name*, *vartype='CONT'*, *lb=-inf*, *ub=inf*, *init=None*, *abstract=False*, *shadow=False*, *key=None*)
Bases: sasoptpy.components.Expression

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

abstract [boolean, optional] Indicator of whether the variable is abstract or not

shadow [boolean, optional] Indicator of whether the variable is shadow or not Used for internal purposes

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')
```

8.3.2 Methods

Variable.set_bounds([lb, ub])	Changes bounds on a variable
Variable.set_init([init])	Changes initial value of a variable

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()
```

8.3.3 Inherited Methods

`Variable.add`(other[, sign])	Combines two expressions and produces a new one
`Variable.copy`([name])	Returns a copy of the `Expression` object
`Variable.get_dual`()	Returns the dual value
`Variable.get_name`()	Returns the name of the expression
`Variable.get_value`()	Returns value of a variable

sasoptpy.Variable.add

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

Combines two expressions and produces a new one

Parameters

other [float or Expression] Second expression or constant value to be added

sign [int, optional] Sign of the addition, 1 or -1

Returns

r [Expression] Reference to the outcome of the operation

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

r [Expression] Copy of the object

Examples

```
>>> x = so.Variable(name='x')
>>> y = so.VariableGroup(1, name='y')
>>> 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

dual [float] Dual value of the variable

sasoptpy.Variable.get_name

`Variable.get_name()`

Returns the name of the expression

Returns

name [string] Name of the expression

Examples

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

sasoptpy.Variable.get_value

`Variable.get_value()`

Returns value of a variable

8.4 Variable Group

8.4.1 Constructor

`VariableGroup`(*argv, name[, vartype, lb, . . .])	Creates a group of `Variable` objects

sasoptpy.VariableGroup

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

Bases: `object`

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')
```

8.4.2 Methods

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

sasoptpy.VariableGroup.get_name

`VariableGroup.`**`get_name`**`()`

Returns the name of the variable group

Returns

name [string] Name of the variable group

Examples

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

sasoptpy.VariableGroup.set_bounds

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

Sets / updates bounds for the given variable

Parameters

lb [float, `pandas.Series`, optional] Lower bound

ub [float, `pandas.Series`, optional] Upper bound

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.set_init

`VariableGroup.set_init(init)`

Sets / updates initial value for the given variable

Parameters

init [float, list, dict, `pandas.Series`] Initial value of the variables

Examples

```
>>> m = so.Model(name='m')
>>> y = m.add_variables(3, name='y')
>>> print(y._defn())
var y {{0,1,2}};
>>> y.set_init(5)
>>> print(y._defn())
var y {{0,1,2}} init 5;
```

sasoptpy.VariableGroup.mult

`VariableGroup.mult(vector)`

Quick multiplication method for the variable groups

Parameters

vector [list, dictionary, `pandas.Series`, or `pandas.DataFrame`] Vector to be multiplied with the variable group

Returns

r [Expression] 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)
```

(continues on next page)

(continued from previous page)

```
>>> 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
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.sum

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

Quick sum method for the variable groups

Parameters

argv [Arguments] List of indices for the sum

Returns

r [Expression] 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']
```

8.5 Constraint

8.5.1 Constructor

Constraint(exp[, direction, name, crange])	Creates a linear or quadratic constraint for optimization models

sasoptpy.Constraint

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

Bases: sasoptpy.components.Expression

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

crange [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

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

```
>>> x = so.Variable(name='x')
>>> y = so.Variable(name='y')
>>> 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')
```

8.5.2 Methods

Constraint.get_value([rhs])	Returns the current value of the constraint
Constraint.set_block(block_number)	Sets the decomposition block number for a constraint
Constraint.set_direction(direction)	Changes the direction of a constraint
Constraint.set_rhs(value)	Changes the RHS of a constraint

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

```
>>> x = so.Variable(name='x', init=2)
>>> c = so.Constraint(x ** 2 + 2 * x <= 15, name='c')
>>> print(c.get_value())
8
>>> print(c.get_value(rhs=True))
-7
```

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_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
```

8.6 Constraint Group

8.6.1 Constructor

`ConstraintGroup`(argv, name)	Creates a group of `Constraint` objects

sasoptpy.ConstraintGroup

class `sasoptpy.ConstraintGroup(argv, name)`

Bases: `object`

Creates a group of `Constraint` objects

Parameters

argv [Generator-type object] A Python generator that includes `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]
]
```

8.6.2 Methods

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

sasoptpy.ConstraintGroup.get_name

ConstraintGroup.**get_name**()

Returns the name of the constraint group

Returns

name [string] Name of the constraint group

Examples

```
>>> m = so.Model(name='m')
>>> x = m.add_variable(name='x')
>>> indices = ['a', 'b', 'c']
```

(continues on next page)

(continued from previous page)

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

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

df [`pandas.DataFrame`] Returns a DataFrame consisting of constraints as expressions

Examples

```
>>> m = so.Model(name='m')
>>> var_ind = ['a', 'b', 'c', 'd']
>>> u = m.add_variables(var_ind, name='u')
>>> t = m.add_variable(name='t')
>>> cg = so.ConstraintGroup((u[i] + 2 * t <= 5 for i in var_ind),
                            name='cg')
>>> ce = cg.get_expressions()
>>> print(ce)
             cg
a   u[a] + 2 * t
b   u[b] + 2 * t
c   u[c] + 2 * t
d   u[d] + 2 * t
>>> ce_rhs = cg.get_expressions(rhs=True)
>>> print(ce_rhs)
                 cg
a   u[a] + 2 * t - 5
b   u[b] + 2 * t - 5
c   u[c] + 2 * t - 5
d   u[d] + 2 * t - 5
```

8.7 Others

8.7.1 Constructors

`ExpressionDict`([name])	Creates a dictionary of `Expression` objects
`ImplicitVar`([argv, name])	Creates an implicit variable
`Set`(name[, init, value, settype])	Creates an index set to be represented inside PROC OPTMODEL

Continued on next page

Table 18 – continued from previous page

SetIterator(initset[, conditions, datatype, ...])	Creates an iterator object for a given Set
Parameter(name[, keys, order, init, value, ...])	Creates a parameter to be represented inside PROC OPTMODEL
ParameterValue(param[, key, prefix, suffix])	Represents a single value of a parameter

sasoptpy.ExpressionDict

class sasoptpy.**ExpressionDict**(*name=None*)

Bases: object

Creates a dictionary of *Expression* objects

Parameters

name [string] Name of the object

Notes

- ExpressionDict is the underlying class for *ImplicitVar*.
- It behaves as a regular dictionary for client-side models.

Examples

```
>>> e[0] = x + 2*y
>>> e[1] = 2*x + y**2
>>> print(e.get_keys())
>>> for i in e:
>>>     print(i, e[i])
(0,) x + 2 * y
(1,) 2 * x + (y) ** (2)
```

sasoptpy.ImplicitVar

class sasoptpy.**ImplicitVar**(*argv=None*, *name=None*)

Bases: sasoptpy.data.ExpressionDict

Creates an implicit variable

Parameters

argv [Generator, optional] Generator object for the implicit variable

name [string, optional] Name of the implicit variable

Notes

- If the loop inside generator is over an abstract object, a definition for the object will be created inside *Model.to_optmodel()* method.

Examples

Regular Implicit Variable

```
>>> I = range(5)
>>> x = so.Variable(name='x')
>>> y = so.VariableGroup(I, name='y')
>>> z = so.ImplicitVar((x + i * y[i] for i in I), name='z')
>>> for i in z:
>>>     print(i, z[i])
(0,) x
(1,) x + y[1]
(2,) x + 2 * y[2]
(3,) x + 3 * y[3]
(4,) x + 4 * y[4]
```

Abstract Implicit Variable

```
>>> I = so.Set(name='I')
>>> x = so.Variable(name='x')
>>> y = so.VariableGroup(I, name='y')
>>> z = so.ImplicitVar((x + i * y[i] for i in I), name='z')
>>> print(z._defn())
impvar z {i_1 in I} = x + i_1 * y[i_1];
>>> for i in z:
>>>     print(i, z[i])
(sasoptpy.data.SetIterator(name=i_1, ...),) x + i_1 * y[i_1]
```

sasoptpy.Set

class sasoptpy.**Set**(*name, init=None, value=None, settype=['num']*)

Bases: sasoptpy.components.Expression

Creates an index set to be represented inside PROC OPTMODEL

Parameters

name [string] Name of the parameter

init [Expression, optional] Initial value expression of the parameter

settype [list, optional] List of types for the set, consisting of 'num' and 'str' values

Examples

```
>>> I = so.Set('I')
>>> print(I._defn())
set I;
```

```
>>> J = so.Set('J', settype=['num', 'str'])
>>> print(J._defn())
set <num, str> J;
```

```
>>> N = so.Parameter(name='N', init=5)
>>> K = so.Set('K', init=so.exp_range(1,N))
```

(continues on next page)

(continued from previous page)

```
>>> print(K._defn())
set K = 1..N;
```

sasoptpy.SetIterator

class sasoptpy.**SetIterator**(*initset*, *conditions=None*, *datatype='num'*, *group={'id': 0, 'order': 1, 'outof': 1}*, *multi_index=False*)

Bases: `sasoptpy.components.Expression`

Creates an iterator object for a given Set

Parameters

initset [Set] Set to be iterated on

conditions [list, optional] List of conditions on the iterator

datatype [string, optional] Type of the iterator

group [dict, optional] Dictionary representing the order of iterator inside multi-index sets

multi_index [boolean, optional] Switch for representing multi-index iterators

Notes

- SetIterator objects are automatically created when looping over a `Set`.
- This class is mainly intended for internal use.
- The `group` parameter consists of following keys
  - **order** : int Order of the parameter inside the group
  - **outof** : int Total number of indices inside the group
  - **id** : int ID number assigned to group by Python

sasoptpy.Parameter

class sasoptpy.**Parameter**(*name*, *keys=None*, *order=1*, *init=None*, *value=None*, *p_type=None*)

Bases: `object`

Creates a parameter to be represented inside PROC OPTMODEL

Parameters

name [string] Name of the parameter

keys [list, optional] List of `Set` to be used as keys for multi-index parameters

init [Expression, optional] Initial value expression of the parameter

p_type [string, optional] Type of the parameter, 'num' or 'str'

See also:

`read_table()`, `Model.read_table()`

Examples

```
>>> p = so.Parameter('p', init=x + 2*y)
>>> print(p._defn())
num p = x + 2 * y;
```

```
>>> I = so.Set('I')
>>> r = so.Parameter('r', keys=I, p_type='str')
>>> print(r._defn())
str r {I};
```

sasoptpy.ParameterValue

class sasoptpy.**ParameterValue**(*param*, *key=None*, *prefix=''*, *suffix=''*)

Bases: `sasoptpy.components.Expression`

Represents a single value of a parameter

Parameters

param [Parameter] Parameter that the value belongs to

key [tuple, optional] Key of the parameter value in the multi-index parameter

prefix [string] Prefix of the parameter

suffix [string] Suffix of the parameter, such as `.lb` and `.ub`

Notes

- Parameter values are mainly used in abstract expressions

8.7.2 Methods

ParameterValue.set_init(val)	Sets the initial value of the parameter

sasoptpy.ParameterValue.set_init

ParameterValue.**set_init**(*val*)

Sets the initial value of the parameter

Parameters

val [Expression] Initial value

Notes

- This method is only available for parameters without index/key.

Examples

```
>>> p = so.Parameter(name='p')
>>> print(p._defn())
num p;
>>> p.set_init(10)
>>> print(p._defn())
num p = 10;
```

8.8 Functions

8.8.1 Utility Functions

check_name(name[, ctype])	Checks if a name is valid and returns a random string if not
dict_to_frame(dictobj[, cols])	Converts dictionaries to DataFrame objects for pretty printing
exp_range(start, stop[, step])	Creates a set within given range
extract_list_value(tuplist, listname)	Extracts values inside various object types
flatten_frame(df[, swap])	Converts a `pandas.DataFrame` object into `pandas.Series`
flatten_tuple(tp)	Flattens nested tuples
get_counter(ctrtype)	Returns and increments the list counter for naming
get_len(i)	Safe wrapper of len() function
get_mutable(exp)	Returns a mutable copy of the given expression if it is immutable
get_namespace()	Prints details of components registered to the global name dictionary
get_solution_table(*argv[, key, 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
list_pack(obj)	Converts a given object to a list
print_model_mps(model)	Prints the MPS representation of the model
quick_sum(argv)	Quick summation function for *Expression* objects
read_data(table, key_set[, key_cols, ...])	(Experimental) Reads a CASTable into PROC OPTMODEL sets
read_frame(df[, cols])	Reads each column in `pandas.DataFrame` into a list of `pandas.Series` objects
read_table(table[, session, key, key_type, ...])	Reads a CAS Table or pandas DataFrame
recursive_walk(obj, func[, attr, alt])	Calls a given method recursively for given objects
register_name(name, obj)	Adds the name and order 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
tuple_unpack(tp)	Grabs the first element in a tuple, if a tuple is given as argument
union(*args)	Returns a union of *Set*, list or set objects
wrap(e[, abstract])	Wraps expression inside another expression

Continued on next page

Table 20 – continued from previous page

utils._evaluate(comp)	Evaluates the value of a given expression component.

sasoptpy.check_name

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

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

Parameters

name [str] Name to be checked if unique

ctype [str, optional] Type of the object

Returns

str [The given name if valid, a random string otherwise]

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

frobj [DataFrame] 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
```

sasoptpy.exp_range

sasoptpy.**exp_range**(*start*, *stop*, *step=1*)

Creates a set within given range

Parameters

start [Expression] First value of the range

stop [Expression] Last value of the range

step [Expression, optional] Step size of the range

Returns

exset [Set] Set that represents the range

Examples

```
>>> N = so.Parameter(name='N')
>>> p = so.exp_range(1, N)
>>> print(p._defn())
set 1..N;
```

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

sasoptpy.flatten_frame

`sasoptpy.flatten_frame(df, swap=False)`

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

Parameters

df [`pandas.DataFrame`] DataFrame to be flattened

swap [boolean, optional] Option to use columns as first index

Returns

new_frame [`pandas.DataFrame`] 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
```

(continues on next page)

(continued from previous page)

```
(coal, period3)       7
(steel, period1)      8
(steel, period2)      4
(steel, period3)      3
(copper, period1)     5
(copper, period2)     7
(copper, period3)     9
dtype: int64
```

sasoptpy.flatten_tuple

`sasoptpy.`**`flatten_tuple`**(*tp*)

Flattens nested tuples

Parameters

tp [tuple] Nested tuple to be flattened

Returns

elem [Generator-type] A generator-type object representing the flat tuple

Examples

```
>>> tp = (3, 4, (5, (1, 0), 2))
>>> print(list(so.flatten_tuple(tp)))
[3, 4, 5, 1, 0, 2]
```

sasoptpy.get_counter

`sasoptpy.`**`get_counter`**(*ctrtype*)

Returns and increments the list counter for naming

Parameters

ctrtype [string] Type of the counter, ‘obj’, ‘var’, ‘con’ or ‘expr’

Returns

ctr [int] Current value of the counter

sasoptpy.get_len

`sasoptpy.`**`get_len`**(*i*)

Safe wrapper of len() function

Returns

leni [int] len(i) if parameter i has len() function defined, otherwise 1

sasoptpy.get_mutable

`sasoptpy.get_mutable(exp)`

Returns a mutable copy of the given expression if it is immutable

Parameters

exp [Variable or Expression] Object to be wrapped

Returns

r [Expression] Mutable copy of the expression, if the original is immutable

sasoptpy.get_namespace

`sasoptpy.get_namespace()`

Prints details of components registered to the global name dictionary

The list includes models, variables, constraints and expressions

Returns

s [string] A string representation of the namespace

sasoptpy.get_solution_table

`sasoptpy.get_solution_table(*argv, key=None, sort=True, rhs=False)`

Returns the requested variable names as a DataFrame table

Parameters

key [list, optional] Keys for objects

sort [bool, optional] Option for sorting the keys

rhs [bool, optional] Option for including constant values

Returns

soltable [`pandas.DataFrame`] DataFrame object that holds keys and values

sasoptpy.list_length

`sasoptpy.list_length(listobj)`

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

Parameters

listobj [list, tuple or dict] Object whose length will be returned

Returns

ln [int] Length of the list, tuple or dict

sasoptpy.list_pack

`sasoptpy.list_pack(obj)`

Converts a given object to a list

If the object is already a list, the function returns the input, otherwise creates a list

Parameters

obj [Object] Object that is converted to a list

Returns

p [list] List that includes the original object

sasoptpy.print_model_mps

`sasoptpy.print_model_mps(model)`

Prints the MPS representation of the model

Parameters

model [Model] Model whose MPS format will be printed

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
```

(continues on next page)

(continued from previous page)

```
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
```

sasoptpy.quick_sum

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

Quick summation function for *Expression* objects

Returns

exp [Expression] 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))
```

sasoptpy.read_data

sasoptpy.**read_data**(*table*, *key_set*, *key_cols=None*, *option=''*, *params=None*)

(Experimental) Reads a CASTable into PROC OPTMODEL sets

Parameters

table [swat.cas.table.CASTable] The CAS table to be read to sets and parameters

key_set [Set] Set object to be read as the key (index)

key_cols [list or string, optional] Column names of the key columns

option [string, optional] Additional options for read data command

params [list, optional] A list of dictionaries where each dictionary represent parameters

Notes

- *key_set* and *key_cols* parameters should be a list. When passing a single item, string type can be used instead.

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] DataFrame to be read

cols [list of strings, optional] Column names to be read. By default, it reads all columns

Returns

series [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
```

sasoptpy.read_table

`sasoptpy.read_table`(*table*, *session=None*, *key=None*, *key_type=None*, *key_name=None*, *columns=None*, *col_types=None*, *col_names=None*, *upload=False*, *casout=None*, *ref=True*)

Reads a CAS Table or pandas DataFrame

Parameters

table [`swat.cas.table.CASTable`, `pandas.DataFrame` object or string] Pointer to CAS Table (server data, CASTable), DataFrame (local data) or the name of the table at execution (server data, string)

session [`swat.CAS` or `saspy.SASsession`] Session object if the table will be uploaded

key [list, optional] List of key columns (for CASTable) or index columns (for DataFrame)

key_type [list or string, optional] A list of column types consists of 'num' or 'str' values

key_name [string, optional] Name of the key set

columns [list, optional] List of columns to read into parameters

col_types [dict, optional] Dictionary of column types

col_names [dict, optional] Dictionary of column names

upload [boolean, optional] Option for uploading a local data to CAS server first

casout [string or dict, optional] Casout options if data is uploaded

ref [boolean, optional] Switch for returning the read data statement generated by the function

Returns

t [tuple] A tuple where first element is the key (index), second element is a list of requested columns and the last element is reference to the original

See also:

`Model.read_table()`, *`Model.read_data()`*

sasoptpy.recursive_walk

`sasoptpy.recursive_walk(obj, func, attr=None, alt=None)`

Calls a given method recursively for given objects

Parameters

func [string] Name of the method / function be called

attr [string, optional] An attribute which triggers an alternative method to be called if exists

alt [string, optional] Name of the alternative method / function to be called if passed attr exists for given objects

Notes

- This function is for internal consumption.

sasoptpy.register_name

`sasoptpy.register_name(name, obj)`

Adds the name and order of a component into the global reference list

Parameters

name [string] Name of the object

obj [object] Object to be registered to the global name dictionary

Returns

objcnt [int] Unique object number to represent creation order

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()
```

(continues on next page)

(continued from previous page)

```
>>> print(so.get_namespace())
Global namespace:
    Model
    VariableGroup
    ConstraintGroup
    Expression
    Variable
    Constraint
```

sasoptpy.tuple_pack

`sasoptpy.tuple_pack(obj)`

Converts a given object to a tuple object

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

Parameters

obj [Object] Object that is converted to a tuple

Returns

t [tuple] Tuple that includes the original object

sasoptpy.tuple_unpack

`sasoptpy.tuple_unpack(tp)`

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

Parameters

tp [tuple]

sasoptpy.union

`sasoptpy.union(*args)`

Returns a union of `Set`, list or set objects

Returns

r [list, set, or Expression] Union of the given arguments

sasoptpy.wrap

`sasoptpy.wrap(e, abstract=False)`

Wraps expression inside another expression

sasoptpy.utils._evaluate

`sasoptpy.utils._evaluate(comp)`

Evaluates the value of a given expression component.

Parameters

comp [dict] Dictionary of references, coefficient and operator

Returns

v [float] Current value of the expression.

8.8.2 Math Functions

math.math_func(exp, op, *args)	Function wrapper for math functions
math.abs(exp)	Absolute value function
math.log(exp)	Natural logarithm function
math.log2(exp)	Logarithm function to the base 2
math.log10(exp)	Logarithm function to the base 10
math.exp(exp)	Exponential function
math.sqrt(exp)	Square root function
math.mod(exp, divisor)	Modulo function
math.int(exp)	Integer value function
math.sign(exp)	Sign value function
math.max(exp, *args)	Largest value function
math.min(exp, *args)	Smallest value function
math.sin(exp)	Sine function
math.cos(exp)	Cosine function
math.tan(exp)	Tangent function

sasoptpy.math.math_func

sasoptpy.math.**math_func**(*exp*, *op*, **args*)

Function wrapper for math functions

Parameters

exp [Expression] Expression where the math func will be applied

op [string] String representation of the math function

args [float, optional] Additional arguments

sasoptpy.math.abs

sasoptpy.math.**abs**(*exp*)

Absolute value function

sasoptpy.math.log

sasoptpy.math.**log**(*exp*)

Natural logarithm function

sasoptpy.math.log2

sasoptpy.math.**log2**(*exp*)

Logarithm function to the base 2

sasoptpy.math.log10

`sasoptpy.math.log10(exp)`
Logarithm function to the base 10

sasoptpy.math.exp

`sasoptpy.math.exp(exp)`
Exponential function

sasoptpy.math.sqrt

`sasoptpy.math.sqrt(exp)`
Square root function

sasoptpy.math.mod

`sasoptpy.math.mod(exp, divisor)`
Modulo function

Parameters

exp [Expression] Dividend

divisor [Expression] Divisor

sasoptpy.math.int

`sasoptpy.math.int(exp)`
Integer value function

sasoptpy.math.sign

`sasoptpy.math.sign(exp)`
Sign value function

sasoptpy.math.max

`sasoptpy.math.max(exp, *args)`
Largest value function

sasoptpy.math.min

`sasoptpy.math.min(exp, *args)`
Smallest value function

sasoptpy.math.sin

`sasoptpy.math.sin(exp)`
Sine function

sasoptpy.math.cos

`sasoptpy.math.`**`cos`**`(`*exp*`)`

Cosine function

sasoptpy.math.tan

`sasoptpy.math.`**`tan`**`(`*exp*`)`

Tangent function

CHAPTER

NINE

RESTFUL API

sasoptpy provides an experimental RESTful API as of v0.2.1. This API is intended to be used to connect development on other languages through a common interface.

Table 1: List of RESTful API requests

Group	Verb	URL	Auth	Action
Overview	GET	/		Returns the server status and version
	POST	/	✓	Performs the requested action
	GET	/workspaces		Returns the list of workspaces
	POST	/workspaces		Creates a new workspace and returns auth token
	GET	/workspaces/{workspace}	✓	Checks if workspace is reachable
	POST	/workspaces/{workspace}		Renews an auth token using initial password
	GET	/sessions	✓	Returns a list of CAS sessions
	POST	/sessions	✓	Creates a new CAS session
Models	GET	/models	✓	Returns a list of models
	POST	/models	✓	Creates a new model
	GET	/models/{model}	✓	Returns the requested model
Variables	GET	/variables	✓	Returns a list of variables
	GET	/models/{model}/variables	✓	Returns a list of variables inside the model
	POST	/models/{model}/variables	✓	Creates a new variable inside the model
	GET	/variable_groups	✓	Returns a list of variable groups
	GET	/models/{model}/variable_groups	✓	Returns a list of variable groups inside the model
	POST	/models/{model}/variable_groups	✓	Creates a set of variable groups inside the model
Expressions	GET	/expressions	✓	Returns a list of expressions
	GET	/models/{model}/objectives	✓	Returns the objective function of a model
	POST	/models/{model}/objectives	✓	Sets the objective function of a model
Constraints	GET	/constraints	✓	Returns a list of constraints
	GET	/models/{model}/constraints	✓	Returns a list of constraints inside the model
	POST	/models/{model}/constraints	✓	Creates a constraint inside the model
	GET	/constraint_groups	✓	Returns a list of constraint groups
	GET	/models/{model}/constraint_groups	✓	Returns a list of constraint groups inside the model
	POST	/models/{model}/constraint_groups	✓	Creates a set of constraint groups inside the model
Solve	POST	/models/{model}/solutions	✓	Solves a model and requests solutions
Data	POST	/data	✓	Creates a data object in the Python server

9.1 Setup

You need to start *sasoptpy*'s web server using

```
from sasoptpy.api import api
api.start()
```

You can also start the web server in a separate thread using *thread=True* parameter.

Status of the server can be checked using a request to the *localhost* address.

9.2 Overview

A *GET* request to the server root returns the package name and current version of the package.

GET /

Get the server status

Status Codes

- 200 OK – Server is available

Example request:

```
GET / HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 49
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "package": "sasoptpy",
  "version": "0.2.1"
}
```

You can perform actions on the server using a *POST* request to the root domain. In order to separate each user's work, an authentication token is required to identify the workspace you are working at. The action you are requesting can only be performed at the workspace level.

POST /

Performs the requested action in a specific workspace

Request Headers

- Authorization – Bearer token to authenticate
- Content-Type – *application/json*

JSON Parameters

- **`action`** (*`string`*) – Action verb (`clean`)

Status Codes

- 200 OK – Action is performed succesfully
- 422 Unprocessable Entity – Authentication is failed

Example request:

```
POST / HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 12
Content-Type: application/x-www-form-urlencoded

action=clean
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 74
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "message": "Cleaned the workspace contents",
  "workspace": "myworkspace"
}
```

You can create a workspace to separate namespace when working on multiple projects at the same time. Each workspace requires a name and a password at initialization. It generates an OAuth2 token to authenticate your upcoming requests. You can see a list of workspaces as follows:

GET /workspaces

Get a list of workspaces

Status Codes

- 200 OK – Action is performed succesfully

Example request:

```
GET /workspaces HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 32
Server: Werkzeug/0.14.1 Python/3.6.5

{
```

(continues on next page)

(continued from previous page)

```
  "workspaces": [
    "myworkspace"
  ]
}
```

A new workspace can be created with a *POST* request as follows. Note that, the `auth` token generated by this request is required for almost all requests you make. This token works as an identifier of the workspace as well, so you do not need to pass the workspace name every time.

POST /workspaces

Creates a new workspace and returns auth token

JSON Parameters

- **name** (*string*) – Workspace name
- **password** (*string*) – Password for token generation and regeneration

Status Codes

- 201 Created – A new workspace is created successfully
- 422 Unprocessable Entity – A workspace with the same name exists

Example request:

```
POST /workspaces HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 31
Content-Type: application/x-www-form-urlencoded

name=myworkspace&password=12345
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 172
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "token": "eyJhbGciOiJIUzI1NiIsImlhdCI6MTU0NjUzODY3OSwiZXhwIjoxNTQ2NTM5Mjc5fQ.
↪eyJuYW1lIjoibXl3b3Jrc3BhY2UifQ.6eNPrOTeK0Nbd42Ch3rPXoYQ558Vj8qL0crTQz8K1_Y",
  "duration": 600
}
```

You can check status of an existing workspace using a *GET* request. This request can also be used to verify that token is valid.

GET /workspaces/(str: *workspace*)

Checks if workspace is reachable

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **workspace** – Name of the workspace

Status Codes

- 200 OK – Token is valid for the workspace
- 422 Unprocessable Entity – Token is not valid for the workspace

Example request:

```
GET /workspaces/myworkspace HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 47
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "status": "Token is valid for the workspace"
}
```

Every token is only valid for a limited time period (currently 10 minutes). You can renew the auth token by sending a *POST* request to target workspace with original password attached.

POST /workspaces/(**str:** *workspace*)

Renews an auth token using initial password

Query Parameters

- **workspace** – Name of the workspace

Form Parameters

- **password** – Password of the workspace

Status Codes

- 201 Created – A new auth token is generated
- 401 Unauthorized – The password does not match the original password

Example request:

```
POST /workspaces/myworkspace HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 14
Content-Type: application/x-www-form-urlencoded

password=12345
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 172
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "token": "eyJhbGciOiJIUzI1NiIsImlhdCI6MTU0NjU0MDQ4MCwiZXhwIjoxNTQ2NTQxMDgwfQ.
↪eyJuYW1lIjoibXl3b3Jrc3BhY2UifQ.cEwUsq6ISz6_T9mAjRJumtJZM5jlOzcOPivRczu1KRE",
  "duration": 600
}
```

A CAS session is required to solve optimization models. You can see a list of CAS sessions in the workspace as follows:

GET /sessions

Returns a list of CAS sessions

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /sessions HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 24
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "sessions": [
    "mycas"
  ]
}
```

You can create a CAS session using a *POST* request. An `authinfo` file on the Python server might be needed to make this connection.

POST /sessions

Creates a new CAS session

Request Headers

- Authorization – Bearer token to authenticate

JSON Parameters

- **name** – CAS session name
- **host** – Hostname of the CAS server
- **port** – Port number used for CAS connections
- **auth** – Absolute path of authinfo file on the Python server

Status Codes

- 201 Created – Created new CAS session
- 400 Bad Request – Cannot create CAS session, see message for more details

Example request:

```
POST /sessions HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 70
Content-Type: application/x-www-form-urlencoded

name=mycas&host=HOST&port=PORT&auth=U%3A%5C.authinfo
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 64
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "id": "8d1767d2-48c8-a048-98c2-3046bb978e6b",
  "name": "mycas"
}
```

9.3 Models

After creating a workspace and a session, you can create multiple models. Models in the same workspace share the same namespace, hence you can use the same model components inside different models.

You can get a list of available models in a workspace using a *GET* request.

GET /models

Returns a list of optimization models

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /models HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 37
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "models": {
    "knapsack": "knapsack"
  }
}
```

A new model can be created using a *POST* request.

POST /models

Creates a new optimization model

Request Headers

- Authorization – Bearer token to authenticate

Form Parameters

- **name** – Model name
- **session** – An existing session name

Status Codes

- 201 Created – Created a new optimization model
- 400 Bad Request – Authentication error

Example request:

```
POST /models HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 27
Content-Type: application/x-www-form-urlencoded

name=knapsack&session=mycas
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 69
```

(continues on next page)

(continued from previous page)

```
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "knapsack",
  "workspace": "myworkspace",
  "session": "mycas"
}
```

Information about an individual model can be requested using a *GET* request, where the model name is a parameter.

GET /models/(str: *model*)
Returns the requested model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not in the workspace

Example request:

```
GET /models/knapsack HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 69
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "knapsack",
  "workspace": "myworkspace",
  "session": "mycas"
}
```

9.4 Variables

Variable objects can be created inside models or standalone. They later can be used in multiple models, if needed. Also, API supports adding multi-dimensional variables using variable groups.

A list of variables in the workspace can be seen using a *GET* request.

GET /variables

Returns a list of variables in the workspace

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /variables HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 34
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "variables": {
    "myvar": "myvar"
  }
}
```

A list of variables that belongs to a certain model can be requested as follows:

GET /models/(str: *model*)/variables

Returns a list of variables inside the model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
GET /models/knapsack/variables HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
```

(continues on next page)

(continued from previous page)

```
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 25
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "variables": [
    "myvar"
  ]
}
```

A variable can be created inside a model using a *POST* request. A variable can be created outside the model scope in *sasoptpy*, but this feature is currently not supported by the RESTful API yet.

POST /models/(str: *model*)/variables

Creates a new variable inside the model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Form Parameters

- **name** – Name of the variable
- **lb** – Lower bound of the variable
- **ub** – Upper bound of the variable
- **vartype** – Type of the variable, *binary integer* or *continuous* (default)

Status Codes

- 201 Created – Variable is created
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
POST /models/knapsack/variables HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 25
Content-Type: application/x-www-form-urlencoded

name=myvar&vartype=binary
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 18
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "myvar"
}
```

Variable groups are multi-dimensional variables inside models. Due to their nature, their construction slightly differs from variables.

A list of variable groups in a workspace can be obtained using a *GET* query.

GET /variable_groups

Returns a list of variable groups

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /variable_groups HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 37
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "variablegroups": {
    "pick": "pick"
  }
}
```

A list of variable groups inside a model can be obtained as follows:

GET /models/(str: *model*)/variable_groups

Returns a list of variable groups inside the model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
GET /models/knapsack/variable_groups HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 29
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "variablegroups": [
    "pick"
  ]
}
```

Creating variable groups requires an additional parameter, called *index*. This parameter should be a JSON array. Each array element can be one of the following: - An integer representing the size, e.g. *4* - A JSON list of strings to be used as index, e.g. *["pen", "watch", "cup"]* - A symbolic data parameter starts with *$* sign, e.g. *"$NumMachines"*

POST /models/(str: *model*)/variable_groups

Creates a set of variable groups inside the model

Request Headers

- Authorization – Bearer token to authenticate
- Content-Length – Length of the JSON object
- Content-Type – *application/json*

Query Parameters

- **model** – Name of the model

JSON Parameters

- **name** – Name of the variable group
- **index** – Set of indices of the variable group
- **lb** – Lower bound of variables
- **ub** – Upper bound of variables
- **vartype** – Variable types
- **init** – Initial value of variables

Status Codes

- 201 Created – Variable group is created
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
POST /models/knapsack/variable_groups HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 74
Content-Type: application/json

{"name": "pick", "index": [["pen", "watch", "cup"]], "vartype": "integer"}
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 57
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "message": "Variable group is created",
  "name": "pick"
}
```

9.5 Expressions

Linear and nonlinear expressions can be added to models to be used inside constraints, objectives, and print statements. Currently, RESTful API only allows adding expressions to be used as objective values.

GET /expressions

Returns a list of expressions in the workspace

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /expressions HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
```

(continues on next page)

(continued from previous page)

```
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 85
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "expressions": {
    "total_value": "5 * pick[pen] + 20 * pick[watch] + 2 * pick[cup]"
  }
}
```

Objective of a model can be obtained using a *GET* request:

GET /models/(str: *model*)/objectives
Returns the objective function of a model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
GET /models/knapsack/objectives HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 83
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "objective": {
    "total_value": "5 * pick[pen] + 20 * pick[watch] + 2 * pick[cup]"
  }
}
```

An objective can be set using a *POST* request. Here, *expression* parameter should conform as a regular Python expression.

POST /models/(str: *model*)/objectives
Sets the objective function of a model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Form Parameters

- **expression** – Objective function as a valid Python expression in string form
- **sense** – Sense of the objective, *minimize* or *maximize*
- **name** – Name of the objective function

Status Codes

- 201 Created – Objective is set
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
POST /models/knapsack/objectives HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 131
Content-Type: application/x-www-form-urlencoded

expression=5*pick["pen"]+20*pick["watch"]+2*pick["cup"]&sense=minimize&name=total_
↪value
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 90
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "total_value",
  "expression": "5 * pick[pen] + 20 * pick[watch] + 2 * pick[cup]"
}
```

9.6 Constraints

Constraint objects are similar to expressions in terms of construction. A list of constraints in the workspace can be requested as follows:

GET /constraints

Returns a list of constraints

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /constraints HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 89
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "constraints": {
    "total_weight": "pick[pen] + 3 * pick[watch] + 10 * pick[cup] <=  22"
  }
}
```

Constraints inside a model can be listed using a *GET* query.

GET /models/(str: *model*)/constraints

Returns a list of constraints inside the model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
GET /models/knapsack/constraints HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
```

(continues on next page)

(continued from previous page)

```
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 34
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "constraints": [
    "total_weight"
  ]
}
```

A single constraint can be created as follows:

POST /models/(str: *model*)/constraints

Creates a constraint inside the model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Form Parameters

- **name** – Name of the constraint
- **expression** – Constraint expression in valid Python format

Status Codes

- 201 Created – Constraint is created
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not found

Example request:

```
POST /models/knapsack/constraints HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 115
Content-Type: application/x-www-form-urlencoded

expression=1*pick["pen"]+3*pick["watch"]+10*pick["cup"]<=22&name=total_weight
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 115
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "total_weight",
  "model": "knapsack",
  "expression": "pick[pen] + 3 * pick[watch] + 10 * pick[cup] <=  22"
}
```

Constraint groups are multi-dimensional constraints. On top of *expression* and *name* parameters, they also require *index* parameter at initialization.

Constraint groups in the workspace can be requested as follows:

GET /constraint_groups

Returns a list of constraint groups

Request Headers

- Authorization – Bearer token to authenticate

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /constraint_groups HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 147
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "constraintgroups": {
    "bounds": {
      "bounds['pen']": "pick[pen] <=  5",
      "bounds['watch']": "pick[watch] <=  5",
      "bounds['cup']": "pick[cup] <=  5"
    }
  }
}
```

Constraint groups that appears inside a model can be requested as follows:

GET /models/(str: *model*)/constraint_groups

Returns a list of constraint groups inside the model

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
GET /models/knapsack/constraint_groups HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 147
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "constraintgroups": {
    "bounds": {
      "bounds['pen']": "pick[pen] <=  5",
      "bounds['watch']": "pick[watch] <=  5",
      "bounds['cup']": "pick[cup] <=  5"
    }
  }
}
```

Constraint group generation is similar to generating variable groups or expressions.

POST /models/(str: *model*)/constraint_groups

Creates a set of constraint groups inside the model

Request Headers

- Authorization – Bearer token to authenticate
- Content-Type – *application/json*

Query Parameters

- **model** – Name of the model

JSON Parameters

- **expression** – Expression in valid Python format
- **index** – Index of the constraint group
- **name** – Name of the constraint group

Status Codes

- 201 Created – Constraint group is created
- 400 Bad Request – Authentication error
- 404 Not Found – Model name is not in the workspace

Example request:

```
POST /models/knapsack/constraint_groups HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 89
Content-Type: application/json

{"expression": "pick[i]<=5", "index": "for i in ['pen','watch','cup']", "name":
↪"bounds"}
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 40
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "bounds",
  "model": "knapsack"
}
```

9.7 Solve

A model can be submitted to be solved by SAS Optimization solvers as follows:

POST /models/(str: *model*)/solutions

Solves a model and requests solutions

Request Headers

- Authorization – Bearer token to authenticate

Query Parameters

- **model** – Name of the model

Form Parameters

- **stream** – *True* for real-time output or *False*

Status Codes

- 200 OK – Valid request with response
- 400 Bad Request – Authentication error

Example request:

```
POST /models/knapsack/solutions HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 12
Content-Type: application/x-www-form-urlencoded

stream=False
```

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 1888
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "model": "knapsack",
  "objective": 125,
  "solutions": {
    "pick[pen]": 5.0,
    "pick[watch]": 5.0,
    "pick[cup]": 0.0
  },
  "stream": "..."
}
```

9.8 Data

Data objects can be created on the server-side using valid Python expressions. This request creates a Python variable with the requested name.

You can test if your request is valid beforehand using:

```
e = json.loads(value)
```

POST /data

Creates a data object in the server

Request Headers

- Authorization – Bearer token to authenticate
- Content-Type – *application/json*

JSON Parameters

- **name** – Name of the object
- **value** – Value in valid JSON format

Status Codes

- 201 Created – Variable is created
- 400 Bad Request – Authentication error

Example request:

```
POST /data HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 22
Content-Type: application/x-www-form-urlencoded

name=capacity&value=20
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 47
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "capacity",
  "type": "int",
  "len": ""
}
```

Example request:

```
POST /data HTTP/1.1
Host: localhost:5000
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 45
Content-Type: application/json

{"name": "value_data", "value": "[5, 20, 2]"}
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 49
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "value_data",
  "type": "list",
  "len": 3
}
```

Example request:

```
POST /data HTTP/1.1
Host: localhost:5000
```

(continues on next page)

(continued from previous page)

```
User-Agent: python-requests/2.19.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Authorization: Bearer eyJhb...K1_Y
Content-Length: 75
Content-Type: application/json

{"name": "weight_data", "value": "{\"pen\": 1, \"watch\": 3, \"cup\": 10}"}
```

Example response:

```
HTTP/1.1 201 CREATED
Content-Type: application/json
Content-Length: 50
Server: Werkzeug/0.14.1 Python/3.6.5

{
  "name": "weight_data",
  "type": "dict",
  "len": 3
}
```

PYTHON MODULE INDEX

S

HTTP ROUTING TABLE

INDEX

Symbols

A

C

D

E

F

G

I

L

M

P

Q

R

S

T

U

V

W