sasoptpy Documentation

Release 1.0.0-beta.2

SAS Institute Inc.

Apr 16, 2020

CONTENTS

PDF Version

Date: Apr 16, 2020 | **Version**: 1.0 | **Release**: 1.0.0-beta.2 | **Reference**: 1.0.0-beta.2-4-g642db765

Links: Repository | Issues | Releases | Community

sasoptpy is a Python package that provides a modeling interface for SAS Optimization and SAS/OR optimization solvers. It provides a quick way for users to deploy optimization models and solve them using SAS Viya and SAS 9.4.

sasoptpy can handle linear, mixed integer linear, nonlinear, and black-box optimization problems. You can use native Python structures such as dictionaries, tuples, and lists to define an optimization problem. sasoptpy offers extensive support of pandas objects.

Under the hood, sasoptpy uses the SAS Scripting Wrapper for Analytic Transfer (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. See SAS Optimization: Mathematical Optimization Procedures for more information about SAS optimization tools.

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

CHAPTER

ONE

OVERVIEW

1.1 About sasoptpy

sasoptpy is a Python package that provides easy and integrated ways of working with optimization solvers in SAS Optimization and SAS/OR. It enables developers to model optimization problems with ease by providing high-level building blocks.

1.1.1 Capabilities

sasoptpy is very flexible in terms of the optimization problem types and workflow alternatives.

Solvers

sasoptpy currently supports the following problem types:

- Linear problems
- Integer linear problems / Mixed integer linear problems
- Quadratic problems
- Nonlinear problems
- Black-box problems

Data

sasoptpy supports working with both client-side data and server-side data. When data are available on the client, it populates the model with integrated data and brings the solution back to the client. When data are available on the server, it generates the code to populate the model on the server. You can retrieve the final solution afterward.

Platforms

sasoptpy can be used with SAS Viya 3.3 or later and SAS 9.4, in all the supported operating systems.

1.1.2 Road Map

The goal of sasoptpy is to support all the functionality of the SAS Optimization and SAS/OR solvers and provide a high-level set of tools for easily working with models.

1.1.3 Versioning

sasoptpy follows Semantic Versioning as of version 1.0.0.

- Any backward incompatible changes increase the major version number (X.y.z).
- Minor changes and improvements increase the the minor version number (x.Y.z).
- Patches increase the patch version number (x.y.Z).
- Pre-releases are marked by using *alpha* and *beta*, and release candidates are marked by using *rc* identifiers.

1.1.4 License

sasoptpy is an open-source package and uses the standard *Apache 2.0 license*.

1.1.5 Support

Have any questions?

- If you have a package-related issue, feel free to report it on GitHub.
- If you have an optimization-related question, consider asking it on SAS Communities.
- For further technical support, contact SAS Technical Support.

1.1.6 Contribution

Contributions are always welcome. Clone the project to your working environment and submit pull requests as you see fit. For more information, see the guidelines at the GitHub repository.

1.1.7 Highlighted Works

A list of highlighted projects and blog posts:

- Fastest, cheapest, greenest: How will football fans choose which matches to attend?
- 1 tournament, 12 countries: A logistical maze?
- Using SAS Optimization with Python and containers
- Bringing Analytics to the Soccer Transfer Season
- Visiting all 30 Major League Baseball Stadiums - with Python and SAS Viya

1.2 What's New

1.2.1 New Features

- Added workspaces; for more information, see *Workspaces in User Guide* and *Efficiency Analysis example*
- Added *package configurations*
- Added *abstract actions* that allow server-side operations. Highlights include:
  - `actions.read_data()` and `actions.create_data()`
  - `actions.for_loop()` and `actions.cofor_loop()`
  - `actions.print_item()`
  - `actions.solve()`
- Added structure decorators for better control of submissions

1.2.2 Changes

- Refactored the entire package; sasoptpy now has *core*, *abstract*, *interface*, *session*, and *util* directories
- Experimental RESTful API was dropped
- *get_obj_by_name* function was removed
- *lso* solver was renamed *blackbox*
- Because of the use of literal strings (**PEP 498**), only Python 3.6 or later versions are supported

1.2.3 Bug Fixes

- Fixed: Arithmetic operations with powers are generating incorrect results
- Fixed: Variable groups with space in their index are not getting values
- Fixed: Constraints without directions do not produce an error
- Fixed: Documentation does not mention conda-forge library requirement
- Fixed: Single-dimensional parameters are hard to access

1.3 License

(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,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
```

(continues on next page)

(continued from previous page)

```
      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
      for use, reproduction, or distribution of Your modifications, or
```

(continues on next page)

(continued from previous page)

```
      for any such Derivative Works as a whole, provided Your use,
      reproduction, and distribution of the Work otherwise complies with
      the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state otherwise,
      any Contribution intentionally submitted for inclusion in the Work
      by You to the Licensor shall be under the terms and conditions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede or modify
      the terms of any separate license agreement you may have executed
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use the trade
      names, trademarks, service marks, or product names of the Licensor,
      except as required for reasonable and customary use in describing the
      origin of the Work and reproducing the content of the NOTICE file.

   7. Disclaimer of Warranty. Unless required by applicable law or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
      PARTICULAR PURPOSE. You are solely responsible for determining the
      appropriateness of using or redistributing the Work and assume any
      risks associated with Your exercise of permissions under this License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and grossly
      negligent acts) or agreed to in writing, shall any Contributor be
      liable to You for damages, including any direct, indirect, special,
      incidental, or consequential damages of any character arising as a
      result of this License or out of the use or inability to use the
      Work (including but not limited to damages for loss of goodwill,
      work stoppage, computer failure or malfunction, or any and all
      other commercial damages or losses), even if such Contributor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.

   END OF TERMS AND CONDITIONS
```

CHAPTER

TWO

INSTALLATION

2.1 Python Version Support and Dependencies

Current version is developed and tested for Python 3.6 and later.

It requires the following packages:

- NumPy
- SASPy
- SWAT
- pandas

2.2 Getting sasoptpy

You can install sasoptpy by using *pip* or *conda*:

```
pip install sasoptpy

conda install -c sas-institute sasoptpy
```

Any dependencies are installed automatically.

Depending on your installation, you might need to add the *conda-forge* channel to *conda*:

```
conda config --append channels conda-forge
```

2.2.1 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
```

CHAPTER

THREE

USER GUIDE

3.1 Introduction to Optimization

Optimization is an umbrella term for maximizing or minimizing a specified target; given as a mathematical function. Optimization is often used in real-life problems from finance to aviation, from chemistry to sports analytics.

Optimization problems can describe a business problem or a physical concept. Any phenomenon that can be represented as a function can be optimized by several algorithms. Optimization lies at the heart of several tools you use every day, from routing to machine learning.

3.1.1 Steps of Optimization

Optimization problems often consist of the following steps:[1][2]

1. Observe the system and define the problem.
2. Gather relevant data.
3. Develop a formulation.
4. Solve the model.
5. Interpret the solution.

A modeler observes the process in order to identify the problems and potential improvements. Several examples of optimization problems are finding the shortest path between two locations, maximizing a profit, and maximizing the accuracy of a handwriting recognition algorithm.

Collecting data is often the most daunting process. In the age of big data, it is often difficult to distinguish noise from relevant data. After data are gathered, you can write a formulation. A proper formulation is critical because features such as linearity and convexity greatly impact the performance of solution algorithms, especially for large problems.

Solving a problem requires an optimization algorithm. SAS Optimization provides several optimization algorithms to solve a variety of different problem types. For more information, see *Types of Optimization*.

Finally, the modeler decides whether the result of an optimization process is valid. If not, the process is repeated by adding the missing pieces until a satisfactory result is obtained.

[1] Hillier, Frederick S., and Gerald J. Lieberman. Introduction to operations research. McGraw-Hill Science, Engineering & Mathematics, 1995.

[2] SAS Institute. SAS/OR 15.1 User's Guide: Mathematical Programming Examples. SAS institute, 2018.

3.1.2 Basic Elements

An optimization formulation has the following elements:

- *Variables* are parameters that the optimization algorithm tunes. An optimization algorithm determines optimal values for variables in the problem. As an example, in an optimization problem of finding a route from your home to the airport, which roads to use are decision variables.
- An *objective* is a performance measure that is to be maximized or minimized. An objective is a function of variables in an optimization problem, meaning an objective value is obtained for specified values of variables. In the home to airport route example, the objective function is the time to reach the airport. The optimization algorithm decides which roads to use in order to minimize the travel time.
- *Constraints* are restrictions on variables that are added in order to prevent illogical or undesired solutions. In the example, the amount of fuel in the car restricts how far you can drive. You can the force optimization algorithm to find a solution under a certain mileage, even if there are other solutions that might be shorter in terms of travel time.

In short, *optimization* consists of choosing *variable* values to maximize or minimize an *objective* function subject to certain *constraints*.

3.1.3 Simple Problem

Consider a simple example. The following problem (brewer's dilemma) is a simplified resource allocation problem, presented by Robert G. Bland.[3]

In the problem, a brewer has limited corn, hops, and barley malt inventory. The brewer wants to produce ale and beer that will maximize the total profit. Each product requires a certain amount of these three ingredients, as follows:

(per barrel)	Amount Required			
Product	Corn (Pounds)	Hops (Ounces)	Barley Malt (Pounds)	Profit ($)
Ale	5	4	35	13
Beer	15	4	20	23
(Total Available)	480	160	1,190	

The variables (*ale* and *beer*) in this problem are the number of barrels of ale and beer to produce. It might be intuitive to prefer beer to ale because of its higher profit rate. However, doing so might deplete all the resources faster and might leave you with an excess amount of hops and barley malt.

The objective in this problem is to maximize the total profit function, which is $13 \times \text{ale} + 23 \times \text{beer}$.

Each limitation on ingredients is a constraint. For corn, hops, and barley malt, the following constrains apply:

$$5 \times \text{ale} + 15 \times \text{beer} \leq 480$$
$$4 \times \text{ale} + 4 \times \text{beer} \leq 160$$
$$35 \times \text{ale} + 20 \times \text{beer} \leq 1,190$$

Combining all items, the optimization formulation is written as follows:

$$\begin{array}{llllll}
\text{maximize:} & 13 \times \text{ale} & +23 \times \text{beer} & & \\
\text{subject to:} & & & & \\
& 5 \times \text{ale} & + 15 \times \text{beer} & \leq & 480 \\
& 4 \times \text{ale} & + 4 \times \text{beer} & \leq & 160 \\
& 35 \times \text{ale} & + 20 \times \text{beer} & \leq & 1,190 \\
& & \text{ale} & \geq & 0 \\
& & \text{beer} & \geq & 0
\end{array}$$

[3] Bland, Robert G. "The Allocation of Resources by Linear Programming." Scientific American 244 (1981): 126-144.

This problem is small enough to be solved by hand, but consider some alternatives.

	Barrels Produced		
#	Ale	Beer	Profit ($)
1	34	0	442
2	0	32	736
3	15	25	770
4	12	28	800

The preceding table indicates that producing only ale or beer creates less profit than producing a combination of the two. Alternative solution 4 gives the optimal values that maximize the profit in this example.

Following are additional examples of problems that can be formulated as optimization problems:

- scheduling project steps to minimize total completion time, where some tasks might depend on completion of earlier tasks
- choosing distribution centers for retailers to minimize total cost while satisfying customer demands on delivery time
- assigning soccer players to a squad to maximize the total rating of the team under foreign player rules
- finding the cheapest travel option and shortest route between two cities
- blending chemical products to minimize the total cost while achieving a certain efficiency of detergents
- choosing a price that will maximize the total profit in a competitive market

For more information about optimization problems and examples, see the related section of SAS Optimization Mathematical Optimization Procedures.

3.1.4 Types of Optimization

The structure of a formulation affects which algorithms can be deployed to solve a problem. The most common optimization types are as follows:

- **Linear optimization**: If the objective function and all constraints of a problem can be described by linear mathematical relations and if all decision variables are continuous, the formulation is called a linear problem (LP). LPs are among the easiest problems in terms of solution time and are well-studied in literature.
- **Mixed integer linear optimization**: If a linear formulation involves binary (yes or no type decisions), or integer variables, the problem is an integer linear problem (ILP) or mixed integer linear problem (MILP), depending on the variables. MILPs are very popular as many real-life problems can be represented as MILPs.
- **Nonlinear optimization**: If a problem involves nonlinear objectives or constraints (such as exponential, polynomial, or absolute values), the problem is called a nonlinear problem (NLP).

3.2 Quick Reference

This is a short introduction to sasoptpy functionality, mainly for new users. You can find more details in the linked chapters.

Using sasoptpy usually consists of the following steps:

1. Create a *CAS session* or a *SAS session*
2. Initialize the *model*

3. Process the *input data*
4. Add the *model components*
5. *Solve the model*

Solving an optimization problem via sasoptpy starts with having a running CAS (SAS Viya) Server or having a SAS 9.4 installation. It is possible to model a problem without a connection but solving a problem requires access to SAS Optimization or SAS/OR solvers at runtime.

3.2.1 Creating a session

Creating a SAS Viya session

To create a SAS Viya session (also called a CAS session), see SWAT documentation. A simple connection can be made using:

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

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

The last two parameters are optional for some use cases.

Creating a SAS 9.4 session

To create a SAS 9.4 session (also called a SAS session), see SASPy documentation. After customizing the configurations for your setup, you can create a session as follows:

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

3.2.2 Initializing a model

After creating a CAS or SAS session, you can create an empty model 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 initializes the optimization model as a `Model` object, called *m*.

3.2.3 Processing input data

The easiest way to work with sasoptpy is to define problem inputs as pandas DataFrames. You can define objective and cost coefficients, and lower and upper bounds by using the DataFrame and Series objects, respectively. See pandas documentation to learn more.

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

In [6]: prob_data = pd.DataFrame([
   ...:     ['Period1', 30, 5],
   ...:     ['Period2', 15, 5],
```

(continues on next page)

(continued from previous page)

```
   ...:     ['Period3', 25, 0]
   ...:     ], columns=['period', 'demand', 'min_prod']).set_index(['period'])
   ...:

In [7]: price_per_product = 10

In [8]: capacity_cost = 10
```

You can refer the set `PERIODS` and the other fields `demand` and `min_production` as follows:

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

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

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

3.2.4 Adding variables

You can add a single variable 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)
  ```

  Then you can add it to an existing model by using *Model.include()*:

  ```
  >>> 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 the input is a set of variables, you can retrieve individual variables by using individual keys, such as `production['Period1']`. To create multidimensional variables, simply list all the keys as follows:

  ```
  >>> multivar = m.add_variables(KEYS1, KEYS2, KEYS3, name='multivar')
  ```

3.2.5 Creating expressions

Expression objects hold 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.

When *Variable* objects are used in a mathematical expression, sasoptpy creates an *Expression* object automatically:

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

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

Note 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 `price_per_product` gives the final expression.

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

3.2.6 Setting an objective function

You can define objective functions in terms of expressions. In this problem, the objective is to maximize the profit, so the `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=
↪'totalProfit')
```

Notice that you can define the same objective by 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` for a minimization problem or `so.MAX` for a maximization problems.

3.2.7 Adding constraints

In sasoptpy, constraints are simply expressions that have 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 is `Model.add_constraint()`, which adds a single constraint to amodel.

The second is `Model.add_constraints()`, which adds multiple constraints 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')
```

The first term, provides a Python generator, which is then 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, respectively. You can define range constraints by using the == symbol followed by a list of two values that represent lower and upper bounds.

```
In [19]: m.add_constraint(production['Period1'] == [10, 100], name='production_bounds
↪')
Out[19]: sasoptpy.Constraint(production[Period1] ==  [10, 100], name='production_
↪bounds')
```

3.2.8 Solving a problem

After a problem is defined, you can send it to the CAS server or SAS session by calling the *Model.solve()* method, which returns the primal solution when it is available, and `None` otherwise.

```
In [20]: 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 8 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 7 linear constraints (6 LE, 0 EQ, 0 GE, 1 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 all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 400.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 4 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 7 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
Out[20]:
Selected Rows from Table SOLUTION

     i                  var  value   lb             ub  rc
0  1.0       production_cap   25.0 -0.0  1.797693e+308 NaN
1  2.0  production[Period1]   25.0  5.0  1.797693e+308 NaN
2  3.0  production[Period2]   15.0  5.0  1.797693e+308 NaN
3  4.0  production[Period3]   25.0 -0.0  1.797693e+308 NaN
```

At the end of the solve operation, the solver returns a "Problem Summary" table and a "Solution Summary" table. These tables can later be accessed by using `m.get_problem_summary()` and `m.get_solution_summary()`.

```
In [21]: print(m.get_solution_summary())
Selected Rows from Table SOLUTIONSUMMARY

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

(continues on next page)

(continued from previous page)

```
Objective Value                    400

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

Best Bound                         400
Nodes                                0
Solutions Found                      3
Iterations                           0
Presolve Time                     0.00
Solution Time                     0.02
```

3.2.9 Printing solutions

You can retrieve the solutions by using the `get_solution_table()` method. It is strongly suggested that you group variables and expressions that share the same keys in a call.

```
In [22]: print(so.get_solution_table(demand, production))
        demand  production
period
Period1     30        25.0
Period2     15        15.0
Period3     25        25.0
```

3.2.10 Initializing a workspace

If you want to use the extensive abstract modeling capabilities of sasoptpy, you can create a workspace. Workspaces support features such as server-side for loops, cofor loops (parallel), reading and creating CAS tables. You can initialize a *Workspace* by using Python's `with` keyword. For example, you can create a workspace that has a set and a variable group as follows:

```
In [23]: def create_workspace():
   ....:     with so.Workspace(name='my_workspace', session=s) as w:
   ....:         I = so.Set(name='I', value=range(1, 11))
   ....:         x = so.VariableGroup(I, name='x', lb=0)
   ....:     return w
   ....:
```

```
In [24]: workspace = create_workspace()

In [25]: print(so.to_optmodel(workspace))
proc optmodel;
   set I = 1..10;
   var x {{I}} >= 0;
quit;
```

You can submit a workspace to a CAS server and retrieve the response by using:

```
In [26]: workspace.submit()
NOTE: Added action set 'optimization'.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 10 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 0 rows and 4 columns.
Out[26]:
Selected Rows from Table SOLUTION

      i    var  value   lb             ub  rc
0   1.0   x[1]    0.0  0.0  1.797693e+308 NaN
1   2.0   x[2]    0.0  0.0  1.797693e+308 NaN
2   3.0   x[3]    0.0  0.0  1.797693e+308 NaN
3   4.0   x[4]    0.0  0.0  1.797693e+308 NaN
4   5.0   x[5]    0.0  0.0  1.797693e+308 NaN
5   6.0   x[6]    0.0  0.0  1.797693e+308 NaN
6   7.0   x[7]    0.0  0.0  1.797693e+308 NaN
7   8.0   x[8]    0.0  0.0  1.797693e+308 NaN
8   9.0   x[9]    0.0  0.0  1.797693e+308 NaN
9  10.0  x[10]    0.0  0.0  1.797693e+308 NaN
```

3.2.11 Package configurations

sasoptpy comes with certain package configurations. The configuration parameters and their default values are as follows:

- verbosity (default 3)
- max_digits (default 12)
- print_digits (default 6)
- default_sense (default so.minimization)
- default_bounds
- valid_outcomes

It is possible to override these configuration parameters. As an example, consider the following constraint representation:

```
In [27]: x = so.Variable(name='x')

In [28]: c = so.Constraint(10 / 3 * x + 1e-20 * x ** 2 <= 30 + 1e-11, name='c')

In [29]: print(so.to_definition(c))
con c : 3.333333333333 * x + 0.0 * ((x) ^ (2)) <= 30.00000000001;
```

You can change the number of digits to be printed as follows:

```
In [30]: so.config['max_digits'] = 2
```

```
In [31]: print(so.to_definition(c))
con c : 3.33 * x + 0.0 * ((x) ^ (2)) <= 30.0;
```

You can remove the maximum number of digits to print as follows:

```
In [32]: so.config['max_digits'] = None
```

```
In [33]: print(so.to_definition(c))
con c : 3.3333333333333335 * x + 1e-20 * ((x) ^ (2)) <= 30.00000000001;
```

You can reset the parameter to its default value by deleting the parameter:

```
In [34]: del so.config['max_digits']
```

You can also create a new configuration to be used globally:

```
In [35]: so.config['myvalue'] = 2
```

3.3 Sessions

3.3.1 CAS Sessions

A `swat.cas.connection.CAS` session is needed in order to use sasoptpy and SAS Optimization solvers to solve optimization problems. You can find more details about CAS sessions in SWAT Documentation.

You can create a sample CAS Session as follows:

```
>>> 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))
```

You can end the session and close the connection as follows:

```
>>> s.terminate()
```

3.3.2 SAS Sessions

A `saspy.SASsession` session is needed in order to use sasoptpy and SAS/OR solvers to solve optimization problems on SAS 9.4 clients.

You can create a sample SAS session as follows:

```
>>> 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'))
```

You can connect to a SAS session by using a configuration file

```
In [1]: sas = saspy.SASsession(cfgfile=config_file)
Using SAS Config named: sshsas
SAS Connection established. Subprocess id is 160
```

```
In [2]: m = so.Model(name='demo', session=sas)
NOTE: Initialized model demo.
```

```
In [3]: print(m.get_session().sasver)
9.04.01M6P11072018
```

You can terminate the SAS session as follows:

```
In [4]: sas.endsas()
SAS Connection terminated. Subprocess id was 160
```

3.4 Models

3.4.1 Creating a model

You can create an empty model by using the `Model` constructor:

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

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

3.4.2 Adding new components to a model

Add 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): [
  ]
]
```

Add 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
  ]
]
```

3.4.3 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)
```

(continues on next page)

(continued from previous page)

```
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, the values of variables are replaced with optimal values.

3.4.4 Accessing components

You can access a list of model variables by using the `Model.get_variables()` method:

```
In [17]: print(m.get_variables())
[sasoptpy.Variable(name='x', lb=0, ub=1, vartype='BIN'), sasoptpy.Variable(name='y',␣
↪lb=1, ub=10, vartype='CONT')]
```

Similarly, you can access a list of constraints by using the `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 by using its name, you can use the `Model.get_constraint()` method:

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

3.4.5 Dropping components

You can drop a variable inside a model by using `Model.drop_variable()` method. Similarly, you can drop a set of variables by using the `Model.drop_variables()` method.

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

In [22]: print(m)
Model: [
  Name: model1
  Objective: MIN [0]
  Variables (1): [
    x
```

(continues on next page)

(continued from previous page)

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

You can drop a constraint by using the *Model.drop_constraint()* method. Similarly, you can drop a set of constraints by using the *Model.drop_constraints()* method.

```
In [23]: m.drop_constraint(c1)

In [24]: m.drop_constraint(c2)

In [25]: print(m)
Model: [
  Name: model1
  Objective: MIN [0]
  Variables (2): [
    x
    y
  ]
  Constraints (0): [
  ]
]
```

```
In [26]: m.include(c1)

In [27]: print(m)
Model: [
  Name: model1
  Objective: MIN [0]
  Variables (2): [
    x
    y
  ]
  Constraints (1): [
    x + 2 * y <=  10
  ]
]
```

3.4.6 Copying a model

You can copy an existing model by including the *Model* object itself.

```
In [28]: copy_model = so.Model(name='copy_model')
NOTE: Initialized model copy_model.

In [29]: copy_model.include(m)

In [30]: print(copy_model)
Model: [
  Name: copy_model
  Objective: MIN [0]
```

(continues on next page)

(continued from previous page)

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

Note that all variables and constraints are included by reference.

3.4.7 Solving a model

A model is solved by using the `Model.solve()` method. This method converts Python definitions into an MPS file and uploads it to a CAS server for the optimization action. The type of the optimization problem is determined according to the variable types and expressions.

```
>>> 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.
```

3.4.8 Solve options

Solver Options

You can pass either solve options from the OPTMODEL procedure or solve parameter from the `solveLp` and `solveMilp` actions by using the `options` parameter of the `Model.solve()` method.

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

The parameter `with` is used to specificy the optimization solver in OPTMODEL procedure. If the `with` parameter is not passed, PROC OPTMODEL chooses a solver that depends on the problem type. Possible `with` values are listed in the SAS/OR documentation.

You can find specific solver options in the SAS Optimization documentation:

- LP solver options
- MILP solver options
- NLP solver options
- QP solver options
- Black-box solver options (formerly called LSO solver)

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

- solveLp options

- solveMilp options

Call parameters

Besides the `options` parameter, you can pass following parameters into the *Model.solve()* method:

- name: Name of the uploaded problem information
- drop: Drops the data from server after the solve
- replace: Replaces an existing data with the same name
- **primalin: Uses the current values of the variables as an initial solution.** When the value of this parameter is `True`, the solve method grabs *Variable* objects' `_init` fields. You can modify this field by using the *Variable.set_init()* method.
- submit: Calls the CAS action or SAS procedure
- frame: Uses the frame (MPS) method. If the value of this parameter is `False`, then the method uses OPTMODEL codes.
- verbose: Prints the generated PROC OPTMODEL code or MPS DataFrame object before the solve

3.4.9 Getting solutions

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

To print the values of any object, you can use the `get_solution_table()` method:

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

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

3.4.10 Tuning MILP model parameters

SAS Optimization solvers provide a variety of settings. However, it might be difficult to find the best settings for a particular model. In order to compare parameters and make a good choice, you can use the *optimization.tune* action for mixed integer linear optimization problems.

The *Model.tune_parameters()* method is a wrapper for the tune action. Consider the following knapsack problem example:

```
In [31]: def get_model():
   ....:     m = so.Model(name='knapsack_with_tuner', session=cas_conn)
   ....:     data = [
   ....:         ['clock', 8, 4, 3],
   ....:         ['mug', 10, 6, 5],
   ....:         ['headphone', 15, 7, 2],
   ....:         ['book', 20, 12, 10],
   ....:         ['pen', 1, 1, 15]
   ....:     ]
   ....:     df = pd.DataFrame(data, columns=['item', 'value', 'weight', 'limit']).set_
↪index(['item'])
   ....:     ITEMS = df.index
```

(continues on next page)

(continued from previous page)

```
   ....:     value = df['value']
   ....:     weight = df['weight']
   ....:     limit = df['limit']
   ....:     total_weight = 55
   ....:     get = m.add_variables(ITEMS, name='get', vartype=so.INT)
   ....:     m.add_constraints((get[i] <= limit[i] for i in ITEMS), name='limit_con')
   ....:     m.add_constraint(so.expr_sum(weight[i] * get[i] for i in ITEMS) <= total_
↪weight, name='weight_con')
   ....:     total_value = so.expr_sum(value[i] * get[i] for i in ITEMS)
   ....:     m.set_objective(total_value, name='total_value', sense=so.MAX)
   ....:     return m
   ....:

In [32]: m = get_model()
NOTE: Initialized model knapsack_with_tuner.
```

For this problem, you can compare configurations as follows:

```
In [33]: results = m.tune_parameters(tunerParameters={'maxConfigs': 10})
NOTE: Added action set 'optimization'.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table KNAPSACK_WITH_
↪TUNER in caslib CASUSER(casuser).
NOTE: The table KNAPSACK_WITH_TUNER has been created in caslib CASUSER(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Start to tune the MILP
         SolveCalls  Configurations    BestTime        Time
                  1               1        0.20        0.25
                  2               2        0.19        0.47
                  3               3        0.18        0.67
                  4               4        0.17        0.86
                  5               5        0.17        1.07
                  6               6        0.17        1.27
                  7               7        0.17        1.48
                  8               8        0.17        1.68
                  9               9        0.17        1.88
                 10              10        0.17        2.09
NOTE: Configuration limit reached.
NOTE: The tuning time is 2.09 seconds.
```

```
In [34]: print(results)
   Configuration conflictSearch  cutGomory cutMiLifted cutStrategy  \
0            0.0      automatic  automatic   automatic   automatic
1            1.0     aggressive       none        none  aggressive
2            2.0       moderate       none   automatic    moderate
3            3.0      automatic       none    moderate   automatic
4            4.0     aggressive       none  aggressive        none
5            5.0       moderate       none   automatic   automatic
6            6.0           none       none    moderate  aggressive
7            7.0      automatic       none        none    moderate
8            8.0      automatic       none    moderate   automatic
9            9.0           none   moderate    moderate        none

  cutZeroHalf heuristics           nodelSel   presolver      probe   restarts  \
0   automatic  automatic          automatic   automatic  automatic  automatic
1    moderate       none              depth   automatic  automatic       none
```

(continues on next page)

(continued from previous page)

```
2  aggressive       none          bestBound    moderate  automatic       none
3    moderate       none  bestEstimatedepth  aggressive      basic   moderate
4    moderate  automatic          bestBound    moderate  automatic       none
5        none  automatic          automatic        none       none      basic
6        none  automatic  bestEstimatedepth    moderate       none      basic
7    moderate  automatic          automatic        none  automatic       none
8    moderate       none  bestEstimatedepth       basic      basic   moderate
9  aggressive       none              depth   automatic  automatic  automatic

     symmetry      varSel  Mean of Run Times  Sum of Run Times  \
0   automatic   automatic               0.19              0.19
1  aggressive   minInfeas               0.17              0.17
2   automatic  ryanFoster               0.17              0.17
3    moderate      pseudo               0.17              0.17
4    moderate  ryanFoster               0.17              0.17
5   automatic      pseudo               0.18              0.18
6        none   automatic               0.18              0.18
7       basic   minInfeas               0.18              0.18
8    moderate      pseudo               0.18              0.18
9        none   minInfeas               0.18              0.18

   Percentage Successful
0                  100.0
1                  100.0
2                  100.0
3                  100.0
4                  100.0
5                  100.0
6                  100.0
7                  100.0
8                  100.0
9                  100.0
```

`Model.tune_parameters()` accepts three main arguments

- milpParameters
- tunerParameters
- tuningParameters

For a full set of tuning parameters and acceptable values of these arguments, see the SAS Optimization documentation.

For the example problem, you can tune the *presolver*, *cutStrategy*, and *strongIter* settings, by using initial values and candidate values, and limit the maximum number of configurations and maximum running time as follows:

```
In [35]: results = m.tune_parameters(
   ....:     milpParameters={'maxtime': 10},
   ....:     tunerParameters={'maxConfigs': 20, 'logfreq': 5},
   ....:     tuningParameters=[
   ....:         {'option': 'presolver', 'initial': 'none', 'values': ['basic',
↪'aggressive', 'none']},
   ....:         {'option': 'cutStrategy'},
   ....:         {'option': 'strongIter', 'initial': -1, 'values': [-1, 100, 1000]}
   ....:     ])
   ....:
NOTE: Added action set 'optimization'.
NOTE: Uploading the problem DataFrame to the server.
```

(continues on next page)

(continued from previous page)

```
NOTE: Cloud Analytic Services made the uploaded file available as table KNAPSACK_WITH_
↪TUNER in caslib CASUSER(casuser).
NOTE: The table KNAPSACK_WITH_TUNER has been created in caslib CASUSER(casuser) from␣
↪binary data uploaded to Cloud Analytic Services.
NOTE: Start to tune the MILP
         SolveCalls  Configurations    BestTime        Time
                  5               5        0.17        1.02
                 10              10        0.17        2.05
                 15              15        0.17        3.04
                 20              20        0.17        4.12
NOTE: Configuration limit reached.
NOTE: The tuning time is 4.12 seconds.
```

```
In [36]: print(results)
    Configuration conflictSearch   cutGomory cutMiLifted cutStrategy  \
0             0.0      automatic   automatic   automatic   automatic
1             1.0       moderate  aggressive  aggressive  aggressive
2             2.0       moderate   automatic  aggressive  aggressive
3             3.0           none    moderate    moderate        none
4             4.0       moderate  aggressive  aggressive  aggressive
5             5.0       moderate        none   automatic    moderate
6             6.0     aggressive        none        none  aggressive
7             7.0       moderate  aggressive  aggressive  aggressive
8             8.0           none        none    moderate  aggressive
9             9.0       moderate  aggressive  aggressive  aggressive
10           10.0       moderate  aggressive  aggressive  aggressive
11           11.0       moderate  aggressive  aggressive  aggressive
12           12.0       moderate        none   automatic   automatic
13           13.0       moderate  aggressive    moderate  aggressive
14           14.0      automatic        none        none    moderate
15           15.0           none    moderate    moderate        none
16           16.0     aggressive        none  aggressive        none
17           17.0       moderate  aggressive  aggressive  aggressive
18           18.0           none    moderate    moderate        none
19           19.0           none    moderate    moderate        none

   cutZeroHalf  heuristics           nodelSel  presolver       probe  \
0    automatic   automatic          automatic  automatic   automatic
1         none    moderate          bestBound   moderate  aggressive
2         none    moderate          automatic   moderate  aggressive
3   aggressive        none              depth  automatic   automatic
4         none    moderate          automatic   moderate  aggressive
5   aggressive        none          bestBound   moderate   automatic
6     moderate        none              depth  automatic   automatic
7   aggressive    moderate          automatic   moderate  aggressive
8         none   automatic  bestEstimatedepth   moderate        none
9         none    moderate          automatic   moderate        none
10        none    moderate          automatic   moderate  aggressive
11        none    moderate          automatic      basic  aggressive
12        none   automatic          automatic       none        none
13        none    moderate          automatic   moderate  aggressive
14    moderate   automatic          automatic       none   automatic
15  aggressive  aggressive              depth  automatic   automatic
16    moderate   automatic          bestBound   moderate   automatic
17        none    moderate              depth   moderate  aggressive
18  aggressive        none              depth  automatic   automatic
```

(continues on next page)

(continued from previous page)

```
19  aggressive        none          automatic  automatic   automatic

     restarts    symmetry      varSel  Mean of Run Times  Sum of Run Times  \
0   automatic   automatic   automatic               0.17              0.17
1        none        none   minInfeas               0.17              0.17
2        none        none   minInfeas               0.17              0.17
3   automatic        none   minInfeas               0.17              0.17
4        none        none   maxInfeas               0.17              0.17
5        none   automatic  ryanFoster               0.17              0.17
6        none  aggressive   minInfeas               0.17              0.17
7        none        none   minInfeas               0.17              0.17
8       basic        none   automatic               0.17              0.17
9        none        none   minInfeas               0.18              0.18
10       none        none   minInfeas               0.18              0.18
11       none        none   minInfeas               0.18              0.18
12      basic   automatic      pseudo               0.18              0.18
13       none        none   minInfeas               0.18              0.18
14       none       basic   minInfeas               0.18              0.18
15  automatic        none   minInfeas               0.18              0.18
16       none    moderate  ryanFoster               0.19              0.19
17       none        none   minInfeas               0.19              0.19
18  automatic   automatic   minInfeas               0.19              0.19
19  automatic        none   minInfeas               0.19              0.19

    Percentage Successful
0                   100.0
1                   100.0
2                   100.0
3                   100.0
4                   100.0
5                   100.0
6                   100.0
7                   100.0
8                   100.0
9                   100.0
10                  100.0
11                  100.0
12                  100.0
13                  100.0
14                  100.0
15                  100.0
16                  100.0
17                  100.0
18                  100.0
19                  100.0
```

You can retrieve full details by using the `Model.get_tuner_results()` method.

3.5 Model Components

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

3.5.1 Expressions

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

Creating expressions

You can create an `Expression` object 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')
```

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 use SAS Viya 3.4 or later or any SAS/OR release for solving your problem.

To use mathematical operations, you need to import *sasoptpy.math* functions.

Mathematical expressions

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

You need to import *sasoptpy.math* into your code to use these functions. Available mathematical functions are listed in *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
```

(continues on next page)

(continued from previous page)

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

Operations

Getting the current value

After the solve is completed, you can obtain the current value of an expression by using the `Expression.get_value()` method:

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

Getting the dual value

You can retrieve the dual values of `Expression` objects by using `Variable.get_dual()` and `Constraint.get_dual()` methods.

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

Addition

You can add, subtract, multiply and divide components using regular Python functionality:

```
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)
```

```
In [15]: share = 0.2 * profit

In [16]: print(share)
sales - 0.6 * material
```

Summation

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

```
In [17]: import time

In [18]: x = m.add_variables(1000, name='x')
```

```
In [19]: t0 = time.time()

In [20]: e = so.expr_sum(2 * x[i] for i in range(1000))

In [21]: print(time.time()-t0)
0.04343891143798828
```

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

In [23]: f = sum(2 * x[i] for i in range(1000))

In [24]: print(time.time()-t0)
1.393463134765625
```

Renaming an expression

You can rename expressions by using the *Expression.set_name()* method:

```
In [25]: e = so.Expression(x[5] + 2 * x[6], name='e1')

In [26]: print(repr(e))
sasoptpy.Expression(exp = x[5] + 2 * x[6], name='e1')
```

```
In [27]: e.set_name('e2');

In [28]: print(repr(e))
sasoptpy.Expression(exp = x[5] + 2 * x[6], name='e2')
```

Copying an expression

You can copy an *Expression* by using the *Expression.copy()* method:

```
In [29]: copy_profit = profit.copy(name='copy_profit')

In [30]: print(repr(copy_profit))
sasoptpy.Expression(exp = 5 * sales - 3 * material, name='copy_profit')
```

3.5.2 Objective Functions

Setting and getting an objective function

You can use any valid *Expression* as the objective function of a model. You can also use an existing expression as an objective function by using the *Model.set_objective()* method. The objective function of a model can be obtained by 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
```

Getting the value

After a solve, you can retrieve the objective value by using the *Model.get_objective_value()* method.

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

3.5.3 Variables

Creating variables

You can create variables either stand-alone 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 a variable is created separately, it needs to be included (or added) inside the model:

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

Equivalently, you could do this in one step:

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

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')
```

Arguments

There are three types of variables: continuous variables, integer variables, and binary variables. Continuous variables are the default type, which you can specify by using the `vartype=so.CONT` argument. You can create integer variables and binary variables by 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.

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')
```

Setting initial values

You can pass the initial values of variables to the solvers for certain problems. The *Variable.set_init()* method changes the initial value for variables. You can set also this value at the creation of the variable.

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

Working with a set of variables

You can create a set of variables by using a single index or by using multiple indices. Valid index sets include list, dict, and `pandas.Index` objects. See *Handling Data* for more information 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')
```

3.5.4 Constraints

Creating constraints

Similar to *Variable* objects, you can create *Constraint* objects 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')
```

Modifying variable coefficients

You can update the coefficient of a variable inside a constraint by 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')
```

Working with a set of constraints

You can add a set of constraints by using a single index or by using multiple indices. Valid index sets include list, dict, and `pandas.Index` objects. See *Handling Data* for more information 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]
]
```

Range constraints

You can give a range for an expression by using a list of two values (lower and upper bound) after 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')
```

3.6 Workspaces

One of the most powerful features of SAS Optimization and PROC OPTMODEL is the ability to combine several optimization models in a single call. You can read a common data set once, or parallelize solve steps for similar subproblems by using this ability.

The newly introduced *Workspace* provides this ability in a familiar syntax. Compared to *Model* objects, a *Workspace* can consist of several models and can use server-side data and OPTMODEL statements in a more detailed way.

You can create several models in the same workspace, and you can solve problems sequentially and concurrently. All the statements are sent to the server after *Workspace.submit()* is called.

3.6.1 Creating a workspace

A *Workspace* should be called by using the `with` Python keyword as follows:

```
>>> with so.Workspace('my_workspace') as w:
>>>     ...
```

3.6.2 Adding components

Unlike *Model* objects, whose components are added explicitly, objects that are defined inside a *Workspace* are added automatically.

For example, adding a new variable is performed as follows:

```
In [1]: with so.Workspace(name='my_workspace') as w:
   ...:     x = so.Variable(name='x', vartype=so.integer)
   ...:
```

You can display contents of a workspace by using the *Workspace.to_optmodel()* method:

```
In [2]: print(w.to_optmodel())
proc optmodel;
   var x integer;
quit;
```

In the following example, data are loaded into the server and a problem is solved by using a workspace:

1. Create CAS session:

```
In [3]: import os

In [4]: hostname = os.getenv('CASHOST')

In [5]: port = os.getenv('CASPORT')

In [6]: from swat import CAS

In [7]: cas_conn = CAS(hostname, port)

In [8]: import sasoptpy as so

In [9]: import pandas as pd
```

2. Upload data:

```
In [10]: def send_data():
   ....:     data = [
   ....:         ['clock', 8, 4, 3],
   ....:         ['mug', 10, 6, 5],
   ....:         ['headphone', 15, 7, 2],
   ....:         ['book', 20, 12, 10],
   ....:         ['pen', 1, 1, 15]
   ....:     ]
   ....:     df = pd.DataFrame(data, columns=['item', 'value', 'weight', 'limit'])
   ....:     cas_conn.upload_frame(df, casout={'name': 'mydata', 'replace': True})
   ....: send_data()
   ....:
NOTE: Cloud Analytic Services made the uploaded file available as table MYDATA in␣
↪caslib CASUSER(casuser).
NOTE: The table MYDATA has been created in caslib CASUSER(casuser) from binary␣
↪data uploaded to Cloud Analytic Services.
```

3. Create workspace and model:

```
In [11]: from sasoptpy.actions import read_data, solve

In [12]: def create_workspace():
   ....:     with so.Workspace('my_knapsack', session=cas_conn) as w:
   ....:         items = so.Set(name='ITEMS', settype=so.string)
   ....:         value = so.ParameterGroup(items, name='value')
   ....:         weight = so.ParameterGroup(items, name='weight')
   ....:         limit = so.ParameterGroup(items, name='limit')
   ....:         total_weight = so.Parameter(name='total_weight', value=55)
   ....:         read_data(
   ....:             table='mydata', index={'target': items, 'key': ['item']},
   ....:             columns=[value, weight, limit]
   ....:         )
   ....:         get = so.VariableGroup(items, name='get', vartype=so.integer, lb=0)
   ....:         limit_con = so.ConstraintGroup((get[i] <= limit[i] for i in items),
   ....:                                        name='limit_con')
   ....:         weight_con = so.Constraint(
   ....:             so.expr_sum(weight[i] * get[i] for i in items) <= total_weight,
   ....:             name='weight_con')
   ....:         total_value = so.Objective(
   ....:             so.expr_sum(value[i] * get[i] for i in items), name='total_value
↪',
   ....:             sense=so.maximize)
   ....:         solve()
   ....:     return w
   ....:

In [13]: my_workspace = create_workspace()
```

4. Print content:

```
In [14]: print(so.to_optmodel(my_workspace))
proc optmodel;
   set <str> ITEMS;
   num value {ITEMS};
   num weight {ITEMS};
   num limit {ITEMS};
```

(continues on next page)

(continued from previous page)

```
   num total_weight = 55;
   read data mydata into ITEMS=[item] value weight limit;
   var get {{ITEMS}} integer >= 0;
   con limit_con {o72 in ITEMS} : get[o72] - limit[o72] <= 0;
   con weight_con : total_weight - (sum {i in ITEMS} (weight[i] * get[i])) >= 0;
   max total_value = sum {i in ITEMS} (value[i] * get[i]);
   solve;
quit;
```

5. Submit:

```
In [15]: my_workspace.submit()
NOTE: Added action set 'optimization'.
NOTE: There were 5 rows read from table 'MYDATA' in caslib 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 (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 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 5 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 6 rows and 4␣
↪columns.
Out[15]:
Selected Rows from Table SOLUTION

     i             var  value   lb             ub   rc
0  1.0       get[book]    2.0 -0.0  1.797693e+308  NaN
1  2.0      get[clock]    3.0 -0.0  1.797693e+308  NaN
2  3.0  get[headphone]    2.0 -0.0  1.797693e+308  NaN
3  4.0        get[mug]   -0.0 -0.0  1.797693e+308  NaN
4  5.0        get[pen]    5.0 -0.0  1.797693e+308  NaN
```

3.6.3 Abstract actions

As shown in the previous example, a *Workspace* can contain statements such as *actions.read_data()* and *actions.solve()*.

These statements are called "Abstract Statements" and are fully supported inside *Workspace* objects. These actions are performed on the server at runtime.

A list of abstract actions is available in the *API section*.

Adding abstract actions

You can import abstract actions through `sasoptpy.actions` as follows:

```
>>> from sasoptpy.actions import read_data, create_data
```

These abstract actions are performed on the server side by generating equivalent OPTMODEL code at execution.

Retrieving results

In order to solve a problem, you need to use the *actions.solve()* function explicitly. Because *Workspace* objects allow several models and solve statements to be included, each of these solve statements is retrieved separately. You can return the solution after each solve by using the `actions.print()` function or by using the *actions.create_data()* function to create table.

In the following example, a parameter is changed and the same problem is solved twice:

1. Create workspace and components:

```
In [16]: from sasoptpy.actions import read_data, solve, print_item

In [17]: def create_multi_solve_workspace():
   ....:     with so.Workspace('my_knapsack', session=cas_conn) as w:
   ....:         items = so.Set(name='ITEMS', settype=so.string)
   ....:         value = so.ParameterGroup(items, name='value')
   ....:         weight = so.ParameterGroup(items, name='weight')
   ....:         limit = so.ParameterGroup(items, name='limit')
   ....:         total_weight = so.Parameter(name='total_weight', init=55)
   ....:         read_data(table='mydata', index={'target': items, 'key': ['item
↪']}, columns=[value, weight, limit])
   ....:         get = so.VariableGroup(items, name='get', vartype=so.integer,␣
↪lb=0)
   ....:         limit_con = so.ConstraintGroup((get[i] <= limit[i] for i in␣
↪items), name='limit_con')
   ....:         weight_con = so.Constraint(
   ....:             so.expr_sum(weight[i] * get[i] for i in items) <= total_
↪weight, name='weight_con')
   ....:         total_value = so.Objective(so.expr_sum(value[i] * get[i] for i␣
↪in items), name='total_value', sense=so.MAX)
   ....:         s1 = solve()
   ....:         p1 = print_item(get)
   ....:         total_weight.set_value(40)
   ....:         s2 = solve()
   ....:         p2 = print_item(get)
   ....:     return w, s1, p1, s2, p2
   ....:
```

(continues on next page)

(continued from previous page)

```
In [18]: (my_workspace, solve1, print1, solve2, print2) = create_multi_solve_
↪workspace()
```

2. Submit to the server:

```
In [19]: my_workspace.submit()
NOTE: Added action set 'optimization'.
NOTE: There were 5 rows read from table 'MYDATA' in caslib 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 (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 8 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: Problem generation will use 8 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 (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 8 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      4     76.0000000    179.0000000   57.54%       0
                0        1      4     76.0000000     77.3333333    1.72%       0
                0        0      4     76.0000000     76.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 76.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 5 rows and 6␣
↪columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 6 rows and 4␣
↪columns.
Out[19]:
Selected Rows from Table SOLUTION

     i              var  value   lb             ub  rc
0  1.0        get[book]    1.0 -0.0  1.797693e+308 NaN
1  2.0       get[clock]    3.0 -0.0  1.797693e+308 NaN
2  3.0   get[headphone]    2.0 -0.0  1.797693e+308 NaN
3  4.0         get[mug]   -0.0 -0.0  1.797693e+308 NaN
4  5.0         get[pen]    2.0 -0.0  1.797693e+308 NaN
```

3. Print results:

```
In [20]: print(solve1.get_solution_summary())
Solution Summary

                                Value
Label
Solver                           MILP
Algorithm              Branch and Cut
Objective Function        total_value
Solution Status               Optimal
Objective Value                    99

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

Best Bound                         99
Nodes                               1
Solutions Found                     4
Iterations                          7
Presolve Time                    0.00
Solution Time                    0.19
```

```
In [21]: print(print1.get_response())
        COL1  get
0       book  2.0
1      clock  3.0
2  headphone  2.0
3        mug -0.0
4        pen  5.0
```

```
In [22]: print(solve2.get_solution_summary())
Solution Summary

                                Value
Label
Solver                           MILP
Algorithm              Branch and Cut
Objective Function        total_value
Solution Status               Optimal
Objective Value                    76
```

(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                         76
Nodes                               1
Solutions Found                     4
Iterations                          3
Presolve Time                    0.00
Solution Time                    0.18
```

```
In [23]: print(print2.get_response())
        COL1  get
0       book  1.0
1      clock  3.0
2  headphone  2.0
3        mug -0.0
4        pen  2.0
```

3.7 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.

3.7.1 Indices

Methods like *`Model.add_variables()`* can use native Python object types such as list and range as variable and constraint indices. You can also use `pandas.Index` objects as indices as well.

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]]
  [Winter: production[Winter]]
  [Spring: production[Spring]]
  [Summer: production[Summer]]
]
```

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

If a list is used as the index set, associated fields such as *lb*, and *ub* should be accessible by using the index keys. Accepted types are dict and `pandas.Series`.

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]', lb=0, ub=1, vartype='BIN')
```

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')
```

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) [
]
```

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

3.7.2 Data

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

pandas DataFrame

`pandas.DataFrame` is the preferred object type for 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) [
  [clock: get[clock]]
  [mug: get[mug]]
  [headphone: get[headphone]]
  [book: get[book]]
  [pen: get[pen]]
]
```

Dictionaries

You can use lists and dictionaries in expressions and when you create variables.

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

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) [
  [clock: get2[clock]]
  [mug: get2[mug]]
  [headphone: get2[headphone]]
  [book: get2[book]]
  [pen: get2[pen]]
]
```

CASTable

When data are available on the server-side, you can pass a reference to the object. Using `swat.cas.table.CASTable` and abstract data requires SAS Viya 3.4 or later.

```
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 TMP8JZH2V6M␣
↪in caslib CASUSER(casuser).
NOTE: The table TMP8JZH2V6M has been created in caslib CASUSER(casuser) from binary␣
↪data uploaded to Cloud Analytic Services.
```

```
In [33]: print(type(table), table)
<class 'swat.cas.table.CASTable'> CASTable('TMP8JZH2V6M', caslib='CASUSER(casuser)')
```

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

In [35]: ITEMS = m.add_set(name='ITEMS')

In [36]: value = m.add_parameter(ITEMS, name='value')

In [37]: weight = m.add_parameter(ITEMS, name='weight')

In [38]: limit = m.add_parameter(ITEMS, name='limit')

In [39]: from sasoptpy.actions import read_data

In [40]: m.include(read_data(table=table, index={'target': ITEMS, 'key': None},
   ....:                          columns=[value, weight, limit]))
   ....:

In [41]: get3 = m2.add_variables(ITEMS, name='get3')

In [42]: print(get3)
```

(continues on next page)

(continued from previous page)

```
Variable Group (get3) [
]
```

Abstract Data

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

```
In [43]: from sasoptpy.actions import read_data

In [44]: m3 = so.Model(name='m3', session=session)
NOTE: Initialized model m3.

In [45]: ITEMS = m.add_set(name='ITEMS')

In [46]: limit = m.add_parameter(ITEMS, name='limit')

In [47]: m3.include(read_data(table='DF', index=['item'], columns=[limit]))

In [48]: print(type(ITEMS), ITEMS)
<class 'sasoptpy.abstract.set.Set'> ITEMS
```

Note that the key set is created as a reference. You can solve the problem later after having the data available with the same name (for example, by using the `swat.cas.connection.CAS.upload_frame()` function)

```
In [49]: session.upload_frame(df, casout='DF')
NOTE: Cloud Analytic Services made the uploaded file available as table DF in caslib␣
↪CASUSER(casuser).
NOTE: The table DF has been created in caslib CASUSER(casuser) from binary data␣
↪uploaded to Cloud Analytic Services.
Out[49]: CASTable('DF', caslib='CASUSER(casuser)')
```

3.7.3 Operations

You can use lists, `pandas.Series`, and `pandas.DataFrame` objects for mathematical operations such as *`VariableGroup.mult()`*.

```
In [50]: sd = [3, 5, 6]

In [51]: z = m.add_variables(3, name='z')
```

```
In [52]: print(z)
Variable Group (z) [
  [0: z[0]]
  [1: z[1]]
  [2: z[2]]
]
```

```
In [53]: print(repr(z))
sasoptpy.VariableGroup([0, 1, 2], name='z')
```

```
In [54]: e1 = z.mult(sd)

In [55]: print(e1)
3 * z[0] + 5 * z[1] + 6 * z[2]
```

```
In [56]: ps = pd.Series(sd)

In [57]: e2 = z.mult(ps)

In [58]: print(e2)
3 * z[0] + 5 * z[1] + 6 * z[2]
```

3.8 Workflows

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

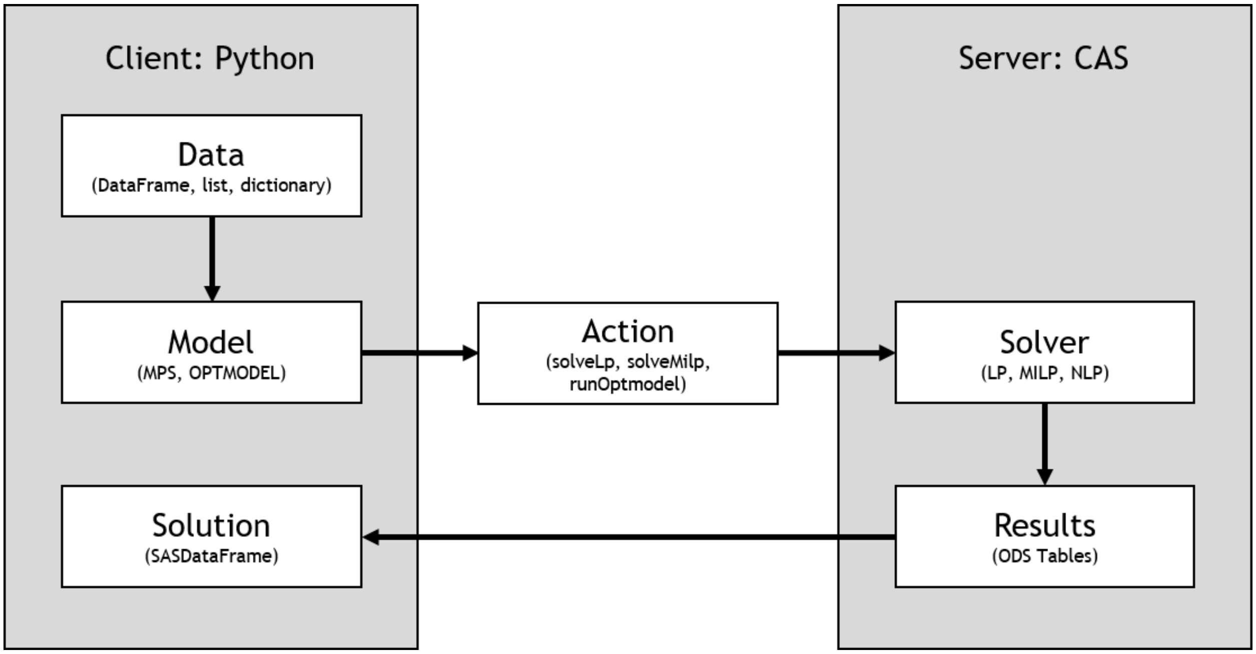
3.8.1 Client-side models

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

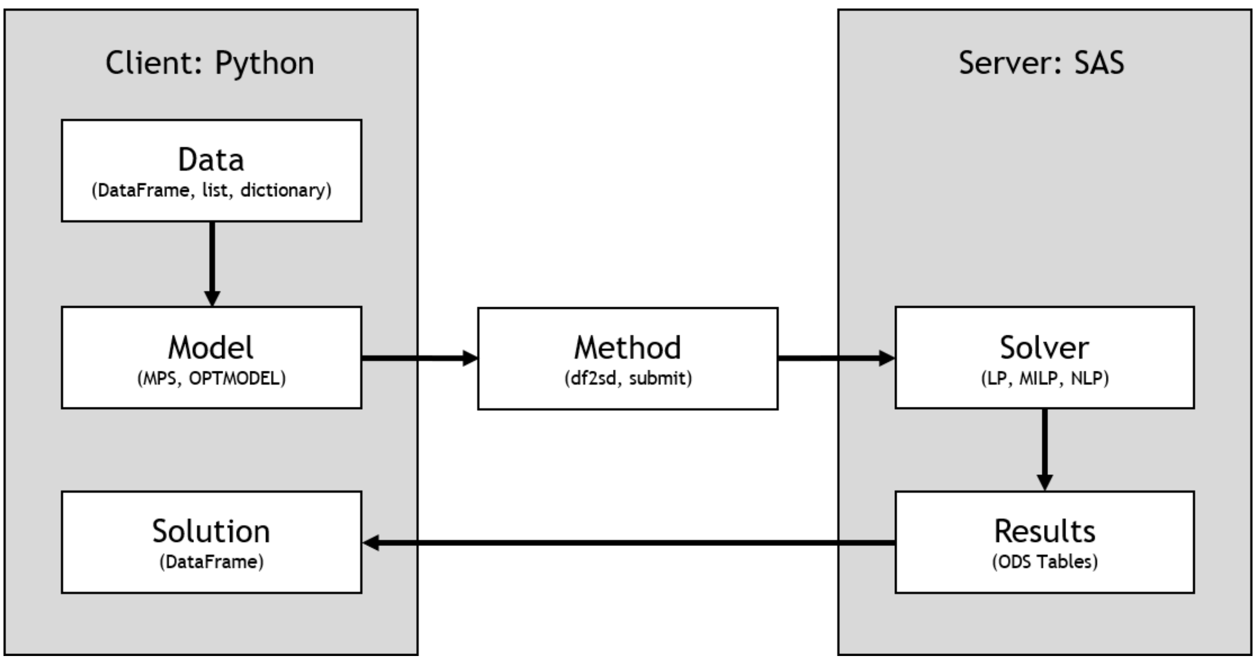
Using a client-side model brings several advantages, such as directly accessing variables, expressions, and constraints. You can more easily perform computationally intensive operations such as filtering data, sorting values, changing variable values, and printing expressions.

There are two main disadvantages of working with client-side models. First, if your model is relatively large, the generated MPS DataFrame or OPTMODEL code might allocate a large amount of memory on your machine. Second, the information that needs to be passed from client to server might be larger than it would be if you use 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.

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)
```

(continues on next page)

(continued from previous page)

```
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 = df.index

In [9]: value = df['value']

In [10]: weight = df['weight']

In [11]: limit = df['limit']

In [12]: total_weight = 55
```

```
In [13]: print(type(ITEMS), ITEMS)
<class 'pandas.core.indexes.range.RangeIndex'> RangeIndex(start=0, stop=5, step=1)
```

```
In [14]: print(type(total_weight), total_weight)
<class 'int'> 55
```

Here, you can obtain the column values one by one:

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

2. Creating the optimization model:

```
# Variables
In [15]: get = m.add_variables(ITEMS, name='get', vartype=so.INT, lb=0)

# Constraints
In [16]: m.add_constraints((get[i] <= limit[i] for i in ITEMS), name='limit_con');

In [17]: m.add_constraint(
   ....:     so.expr_sum(weight[i] * get[i] for i in ITEMS) <= total_weight,
   ....:     name='weight_con');
   ....:

# Objective
In [18]: total_value = so.expr_sum(value[i] * get[i] for i in ITEMS)

In [19]: m.set_objective(total_value, name='total_value', sense=so.MAX);
```

(continues on next page)

(continued from previous page)

```
# Solve
In [20]: m.solve(verbose=True)
NOTE: Added action set 'optimization'.
NOTE: Converting model client_CAS to OPTMODEL.
   var get {{0,1,2,3,4}} integer >= 0;
   con limit_con_0 : get[0] <= 3;
   con limit_con_1 : get[1] <= 5;
   con limit_con_2 : get[2] <= 2;
   con limit_con_3 : get[3] <= 10;
   con limit_con_4 : get[4] <= 15;
   con weight_con : 4 * get[0] + 6 * get[1] + 7 * get[2] + 12 * get[3] + get[4]
→<= 55;
   max total_value = 8 * get[0] + 10 * get[1] + 15 * get[2] + 20 * get[3] +␣
→get[4];
   solve;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.
→lb ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_
→CON_.dual;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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).
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 8 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%       0
NOTE: Optimal.
NOTE: Objective = 99.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 5 rows and 6␣
→columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 6 rows and 4␣
→columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows␣
→and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and␣
→4 columns.
Out[20]:
Selected Rows from Table SOLUTION
```

(continues on next page)

(continued from previous page)

```
     i     var  value   lb             ub  rc
0  1.0  get[0]    2.0 -0.0  1.797693e+308 NaN
1  2.0  get[1]    5.0 -0.0  1.797693e+308 NaN
2  3.0  get[2]    2.0 -0.0  1.797693e+308 NaN
3  4.0  get[3]   -0.0 -0.0  1.797693e+308 NaN
4  5.0  get[4]    3.0 -0.0  1.797693e+308 NaN
```

You can display the generated OPTMODEL code at run time by using the `verbose=True` option. Here, you can see the coefficient values of the parameters inside the model.

3. Parsing the results:

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

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

```
In [21]: print(m.get_solution())
Selected Rows from Table SOLUTION

     i     var  value   lb             ub  rc
0  1.0  get[0]    2.0 -0.0  1.797693e+308 NaN
1  2.0  get[1]    5.0 -0.0  1.797693e+308 NaN
2  3.0  get[2]    2.0 -0.0  1.797693e+308 NaN
3  4.0  get[3]   -0.0 -0.0  1.797693e+308 NaN
4  5.0  get[4]    3.0 -0.0  1.797693e+308 NaN
```

```
In [22]: print(so.get_solution_table(get, key=ITEMS))
   get
0  2.0
1  5.0
2  2.0
3 -0.0
4  3.0
```

```
In [23]: print('Total value:', total_value.get_value())
Total value: 99.0
```

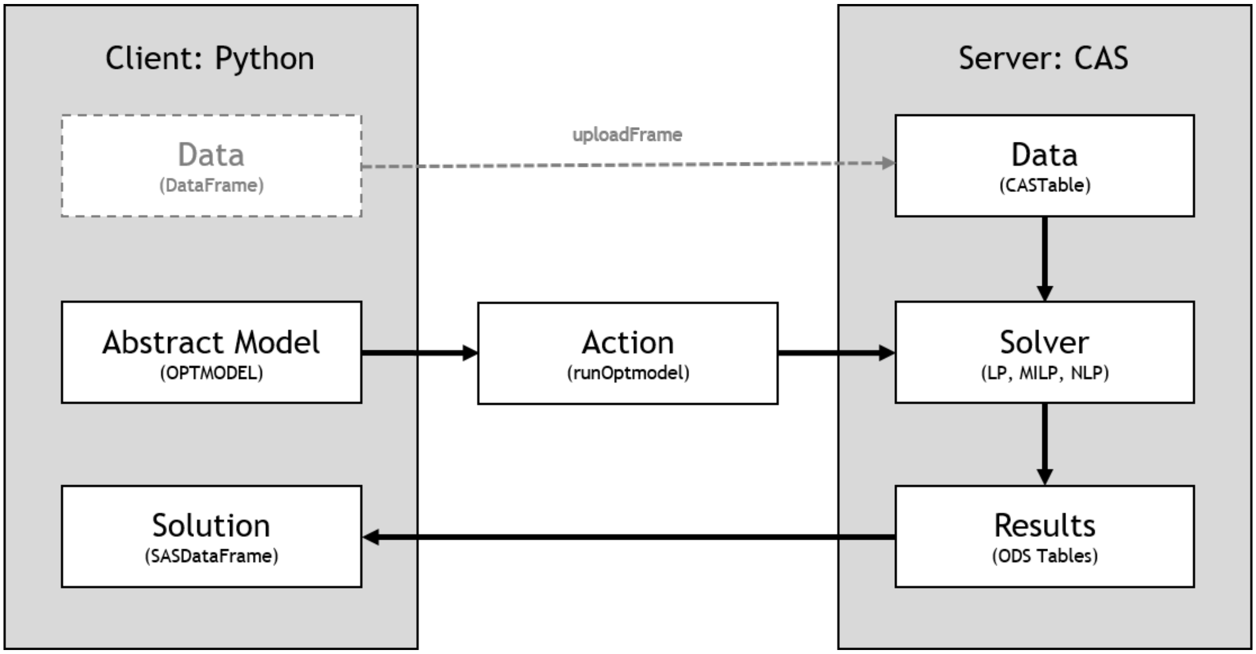
3.8.2 Server-side models

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

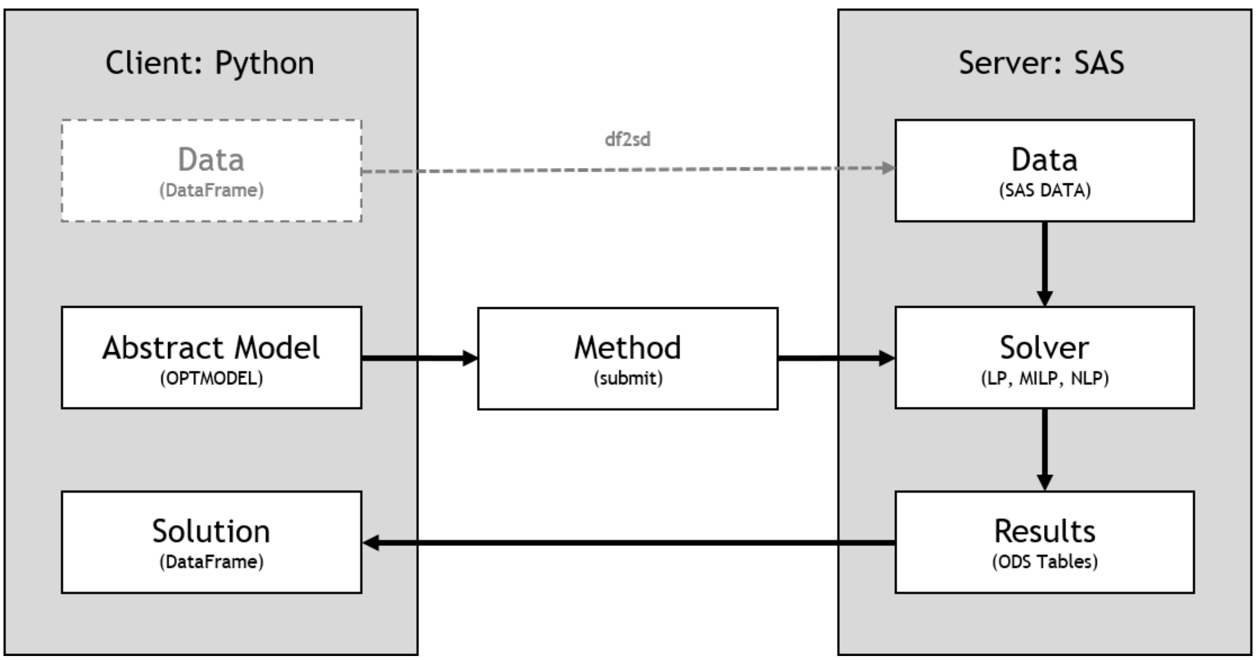
The main advantage of the server-side models is faster upload times compared to client-side models. This is especially noticeable when using large numbers of variable and constraint groups.

The only disadvantage of using server-side models is that variables often need to be accessed directly from the resulting SASDataFrame objects. Because 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 steps, the same example is solved by using server-side data.

1. Creating the optimization model:

```
In [24]: from sasoptpy.actions import read_data

In [25]: m = so.Model(name='client_CAS', session=session)
NOTE: Initialized model client_CAS.
```

```
In [26]: cas_table = session.upload_frame(df, casout='data')
NOTE: Cloud Analytic Services made the uploaded file available as table DATA in
↪caslib CASUSER(casuser).
```

(continues on next page)

(continued from previous page)

```
NOTE: The table DATA has been created in caslib CASUSER(casuser) from binary data␣
↪uploaded to Cloud Analytic Services.
```

```
In [27]: ITEMS = m.add_set(name='ITEMS', settype=so.STR)

In [28]: value = m.add_parameter(ITEMS, name='value')

In [29]: weight = m.add_parameter(ITEMS, name='weight')

In [30]: limit = m.add_parameter(ITEMS, name='limit')

In [31]: m.include(read_data(
   ....:    table=cas_table, index={'target':ITEMS, 'key': 'item'},
   ....:    columns=[value, weight, limit]))
   ....:

# Variables
In [32]: get = m.add_variables(ITEMS, name='get', vartype=so.INT, lb=0)

# Constraints
In [33]: m.add_constraints((get[i] <= limit[i] for i in ITEMS), name='limit_con');

In [34]: m.add_constraint(
   ....:     so.expr_sum(weight[i] * get[i] for i in ITEMS) <= total_weight,
   ....:     name='weight_con');
   ....:
```

```
# Objective
In [35]: total_value = so.expr_sum(value[i] * get[i] for i in ITEMS)

In [36]: m.set_objective(total_value, name='total_value', sense=so.MAX);

# Solve
In [37]: m.solve(verbose=True)
NOTE: Added action set 'optimization'.
NOTE: Converting model client_CAS to OPTMODEL.
   set <str> ITEMS;
   num value {{ITEMS}};
   num weight {{ITEMS}};
   num limit {{ITEMS}};
   read data DATA into ITEMS=[item] value weight limit;
   var get {{ITEMS}} integer >= 0;
   con limit_con {o37 in ITEMS} : get[o37] - limit[o37] <= 0;
   con weight_con : sum {i in ITEMS} (weight[i] * get[i]) <= 55;
   max total_value = sum {i in ITEMS} (value[i] * get[i]);
   solve;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.
↪lb ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_
↪CON_.dual;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: There were 5 rows read from table 'DATA' in caslib 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 (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).
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 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 5 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 6 rows and 4␣
↪columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows␣
↪and 4 columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and␣
↪4 columns.
Out[37]:
Selected Rows from Table SOLUTION

     i             var  value   lb             ub  rc
0  1.0       get[book]    2.0 -0.0  1.797693e+308 NaN
1  2.0      get[clock]    3.0 -0.0  1.797693e+308 NaN
2  3.0  get[headphone]    2.0 -0.0  1.797693e+308 NaN
3  4.0        get[mug]   -0.0 -0.0  1.797693e+308 NaN
4  5.0        get[pen]    5.0 -0.0  1.797693e+308 NaN
```

2. Parsing the results:

```
# Print results
In [38]: print(m.get_solution())
Selected Rows from Table SOLUTION

     i             var  value   lb             ub  rc
0  1.0       get[book]    2.0 -0.0  1.797693e+308 NaN
1  2.0      get[clock]    3.0 -0.0  1.797693e+308 NaN
2  3.0  get[headphone]    2.0 -0.0  1.797693e+308 NaN
3  4.0        get[mug]   -0.0 -0.0  1.797693e+308 NaN
4  5.0        get[pen]    5.0 -0.0  1.797693e+308 NaN
```

```
In [39]: print('Total value:', m.get_objective_value())
Total value: 99.0
```

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

3.8.3 Limitations

- In SAS Viya, nonlinear models can be solved only by using the runOptmodel action, which requires SAS Viya 3.4 or later.
- User-defined decomposition blocks are available only in MPS mode, and therefore work only with client-side data.
- Mixed usage (client-side and server-side data) might not work in some cases. A quick fix would be transferring the data in either direction.

CHAPTER

FOUR

EXAMPLES

Examples are provided from SAS/OR documentation.

This chapter is split into 3 parts.

- The *first part* of examples demonstrate using SAS Viya (SAS Optimization) solvers with concrete problem formulation.
- The *second part* of examples demonstrate using SAS Viya (SAS Optimization) solvers with abstract problem formulation.
- The *third part* of examples demonstrate using SAS 9.4 (SAS/OR) solvers.

4.1 SAS Viya Examples (Concrete)

4.1.1 Food Manufacture 1

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex1_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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]]
```

(continues on next page)

(continued from previous page)

```
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
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.expr_sum(revenue_per_ton * manufacture[p] for p in PERIODS)
rawcost = so.expr_sum(cost.at[o, p] * buy[o, p]
                       for o in OILS for p in PERIODS)
storagecost = so.expr_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.expr_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.expr_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'})
```

(continues on next page)

(continued from previous page)

```
    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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.food_manufacture_1 import test

In [8]: 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 8 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         49    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 54 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 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 8 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    1.749040E+03         0
          P 2         32    3.638889E+04         0
          D 2         51    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The Primal Simplex solve time is 0.01 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 54 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 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 8 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 8 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
```

(continues on next page)

(continued from previous page)

```
            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 0.01 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    1.014226E+03         0
          D C         13    9.697429E+00         0
          D 2         16    1.078426E+05         0
          P 2         17    1.078426E+05         0
          D 2         18    1.078426E+05         0
NOTE: The Crossover time is 0.01 seconds.
NOTE: Optimal.
NOTE: Objective = 107842.59259.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 54 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 16 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 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 8 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 20 rows (40.00%), 29 columns (56.86%), and 1 component.
NOTE: The network extraction and setup time is 0.01 seconds.
                           Primal         Primal           Dual
         Iteration      Objective  Infeasibility  Infeasibility     Time
                 1   3.750000E+03   5.000000E+02   1.551000E+03     0.01
                24   7.125000E+04   0.000000E+00   0.000000E+00     0.01
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.01 seconds.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    2.240180E+05         0
          P 2         43    1.078426E+05         0
NOTE: Optimal.
NOTE: Objective = 107842.59259.
```

(continues on next page)

(continued from previous page)

```
NOTE: The Simplex solve time is 0.02 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 95 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 54 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 14 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
                 buy           use         store
veg1 1  0.000000e+00  8.518519e+01  4.148148e+02
veg1 2  0.000000e+00  1.592593e+02  2.555556e+02
veg1 3  2.842171e-14  0.000000e+00  2.555556e+02
veg1 4 -1.421085e-14  1.592593e+02  9.629630e+01
veg1 5  7.105427e-14  9.629630e+01  0.000000e+00
veg1 6  6.592593e+02  1.592593e+02  5.000000e+02
veg2 1 -5.684342e-14  1.148148e+02  3.851852e+02
veg2 2  0.000000e+00  4.074074e+01  3.444444e+02
veg2 3  2.842171e-14  2.000000e+02  1.444444e+02
veg2 4 -2.842171e-14  4.074074e+01  1.037037e+02
veg2 5  0.000000e+00  1.037037e+02  0.000000e+00
veg2 6  5.407407e+02  4.074074e+01  5.000000e+02
oil1 1  0.000000e+00  0.000000e+00  5.000000e+02
oil1 2  0.000000e+00  0.000000e+00  5.000000e+02
oil1 3  0.000000e+00  0.000000e+00  5.000000e+02
oil1 4  0.000000e+00 -1.744059e-14  5.000000e+02
oil1 5  0.000000e+00  0.000000e+00  5.000000e+02
oil1 6  0.000000e+00  0.000000e+00  5.000000e+02
oil2 1  0.000000e+00  0.000000e+00  5.000000e+02
oil2 2  2.500000e+02  2.500000e+02  5.000000e+02
oil2 3  0.000000e+00  2.273737e-13  5.000000e+02
oil2 4  2.842171e-14  2.500000e+02  2.500000e+02
oil2 5  0.000000e+00  2.500000e+02  0.000000e+00
oil2 6  7.500000e+02  2.500000e+02  5.000000e+02
oil3 1  0.000000e+00  2.500000e+02  2.500000e+02
oil3 2  0.000000e+00  0.000000e+00  2.500000e+02
oil3 3 -5.048710e-29  2.500000e+02 -2.842171e-13
oil3 4  2.842171e-13  0.000000e+00  0.000000e+00
oil3 5  5.000000e+02  0.000000e+00  5.000000e+02
oil3 6  0.000000e+00  0.000000e+00  5.000000e+02
veg1 0           NaN           NaN  5.000000e+02
veg2 0           NaN           NaN  5.000000e+02
oil1 0           NaN           NaN  5.000000e+02
oil2 0           NaN           NaN  5.000000e+02
oil3 0           NaN           NaN  5.000000e+02
Out[8]: 107842.59259259264
```

4.1.2 Food Manufacture 2

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex2_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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))}

    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)
```

(continues on next page)

(continued from previous page)

```
    VEG = [i for i in OILS if 'veg' in i]
    NONVEG = [i for i in OILS if i not in VEG]
    revenue = so.expr_sum(revenue_per_ton * manufacture[p] for p in PERIODS)
    rawcost = so.expr_sum(cost.at[o, p] * buy[o, p]
                          for o in OILS for p in PERIODS)
    storagecost = so.expr_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.expr_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.expr_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)
    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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.food_manufacture_2 import test

In [8]: 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 8 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 8 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      5  36900.0000000         343250   89.25%       0
                0        1      5  36900.0000000         107333   65.62%       0
                0        1      5  36900.0000000         105799   65.12%       0
                0        1      5  36900.0000000         105650   65.07%       0
                0        1      5  36900.0000000         105650   65.07%       0
                0        1      5  36900.0000000         105650   65.07%       0
                0        1      5  36900.0000000         105650   65.07%       0
                0        1      5  36900.0000000         105650   65.07%       0
                0        1      6  99491.6666667         105650    5.83%       0
NOTE: The MILP solver added 15 cuts with 77 cut coefficients at the root.
               10        6      7  99683.3343492         105090    5.15%       0
               28       19      8  99908.3333333         104782    4.65%       0
               68       45      9  99908.3333333         104564    4.45%       0
              139       80     10         100054         104225    4.00%       0
              145       75     11         100192         103683    3.37%       0
              177       86     12         100192         103516    3.21%       0
              183       87     13         100214         103516    3.19%       0
              189       85     14         100279         103268    2.89%       0
```

(continues on next page)

(continued from previous page)

```
             283       40      15          100279          102053     1.74%        0
             284       40      16          100279          102053     1.74%        0
             333        0      16          100279          100279     0.00%        0
NOTE: Optimal.
NOTE: Objective = 100278.70577.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 125 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 132 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
                 buy           use         store       is_used
veg1 1  0.000000e+00  8.518519e+01  4.148148e+02  1.000000e+00
veg1 2  0.000000e+00  8.518519e+01  3.296296e+02  1.000000e+00
veg1 3  0.000000e+00  0.000000e+00  3.296296e+02  0.000000e+00
veg1 4  0.000000e+00  1.550000e+02  1.746296e+02  9.999948e-01
veg1 5  0.000000e+00  1.550000e+02  1.962960e+01  1.000000e+00
veg1 6  4.803704e+02  0.000000e+00  5.000000e+02  0.000000e+00
veg2 1  0.000000e+00  1.148148e+02  3.851852e+02  1.000000e+00
veg2 2  0.000000e+00  1.148148e+02  2.703704e+02  1.000000e+00
veg2 3  0.000000e+00  2.000000e+02  7.037037e+01  1.000000e+00
veg2 4 -1.421085e-14  0.000000e+00  7.037037e+01 -1.949310e-15
veg2 5  0.000000e+00  0.000000e+00  7.037037e+01 -0.000000e+00
veg2 6  6.296296e+02  2.000000e+02  5.000000e+02  1.000000e+00
oil1 1  0.000000e+00  0.000000e+00  5.000000e+02  0.000000e+00
oil1 2 -5.684342e-14  0.000000e+00  5.000000e+02  0.000000e+00
oil1 3  0.000000e+00  0.000000e+00  5.000000e+02 -0.000000e+00
oil1 4  5.684342e-14  0.000000e+00  5.000000e+02  0.000000e+00
oil1 5  0.000000e+00  0.000000e+00  5.000000e+02  0.000000e+00
oil1 6  0.000000e+00  0.000000e+00  5.000000e+02  0.000000e+00
oil2 1  0.000000e+00  0.000000e+00  5.000000e+02 -0.000000e+00
oil2 2  1.900001e+02  2.273737e-13  6.900001e+02  9.094947e-16
oil2 3  0.000000e+00  2.300000e+02  4.600001e+02  1.000000e+00
oil2 4 -2.842171e-14  2.300001e+02  2.300000e+02  1.000000e+00
oil2 5  2.842171e-14  2.300000e+02  0.000000e+00  1.000000e+00
oil2 6  7.300000e+02  2.300000e+02  5.000000e+02  1.000000e+00
oil3 1  0.000000e+00  2.500000e+02  2.500000e+02  1.000000e+00
oil3 2  0.000000e+00  2.500000e+02  2.557954e-13  1.000000e+00
oil3 3  5.799999e+02  2.000000e+01  5.599999e+02  1.000000e+00
oil3 4  0.000000e+00  1.999990e+01  5.400000e+02  9.999948e-01
oil3 5  0.000000e+00  2.000000e+01  5.200000e+02  1.000000e+00
oil3 6  0.000000e+00  2.000000e+01  5.000000e+02  1.000000e+00
veg1 0           NaN           NaN  5.000000e+02           NaN
veg2 0           NaN           NaN  5.000000e+02           NaN
oil1 0           NaN           NaN  5.000000e+02           NaN
oil2 0           NaN           NaN  5.000000e+02           NaN
oil3 0           NaN           NaN  5.000000e+02           NaN
Out[8]: 100278.70576513262
```

4.1.3 Factory Planning 1

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex3_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: https://support.sas.com/documentation/onlinedoc/or/ex_code/151/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.expr_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.expr_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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.factory_planning_1 import test

In [8]: test(cas_conn)
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 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 126 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 72 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
                   make        sell  store
(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
```

(continues on next page)

(continued from previous page)

```
(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
(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[8]: 93715.17857142858
```

4.1.4 Factory Planning 2

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex4_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/mpex04.html

Model

```
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))
    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(),
```

(continues on next page)

(continued from previous page)

```
                             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.expr_sum(
        storage_cost_per_unit * store[p, t] for p in PRODUCTS for t in PERIODS)
    revenue = so.expr_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.expr_sum(production_time.at[mc, p] * make[p, t] for p in PRODUCTS)
        <= num_hours_per_period *
        (num_machines[mc] - numMachinesDown[mc, t])
        for mc in MACHINE_TYPES for t in PERIODS), name='machine_hours_con')

    m.add_constraints((so.expr_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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)
```

(continues on next page)

(continued from previous page)

```
In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.factory_planning_2 import test

In [8]: 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 8 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 8 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
                0        1      2  92755.0000000         111392   16.73%       0
                0        1      2  92755.0000000         111136   16.54%       0
                0        1      2  92755.0000000         110056   15.72%       0
                0        1      2  92755.0000000         109718   15.46%       0
                0        1      2  92755.0000000         109122   15.00%       0
                0        1      2  92755.0000000         108904   14.83%       0
                0        1      2  92755.0000000         108868   14.80%       0
                0        1      2  92755.0000000         108855   14.79%       0
                0        1      3         108855         108855    0.00%       0
NOTE: The MILP solver added 38 cuts with 136 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 108855.00961.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 156 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 77 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
                  make         sell       store
(prod1, 1)   500.000000   500.000000   0.000000
```

(continues on next page)

(continued from previous page)

```
(prod1, 2)   600.000200   600.000000    0.000200
(prod1, 3)   399.999201   300.000000   99.999401
(prod1, 4)     0.001198   100.000599    0.000000
(prod1, 5)     0.000000     0.000000    0.000000
(prod1, 6)   550.000000   500.000000   50.000000
(prod2, 1)  1000.000000  1000.000000    0.000000
(prod2, 2)   500.000188   500.000000    0.000188
(prod2, 3)   699.998215   599.999002   99.999401
(prod2, 4)     0.001797   100.001198    0.000000
(prod2, 5)   100.002196   100.000000    0.002196
(prod2, 6)   549.997804   500.000000   50.000000
(prod3, 1)   300.000000   300.000000    0.000000
(prod3, 2)   200.000000   200.000000    0.000000
(prod3, 3)   100.000000     0.000000  100.000000
(prod3, 4)     0.001283   100.001283    0.000000
(prod3, 5)   500.000072   500.000000    0.000072
(prod3, 6)   149.999928   100.000000   50.000000
(prod4, 1)   300.000000   300.000000    0.000000
(prod4, 2)     0.000000     0.000000    0.000000
(prod4, 3)    99.999401     0.000000   99.999401
(prod4, 4)     0.002994   100.002396    0.000000
(prod4, 5)   100.000000   100.000000    0.000000
(prod4, 6)   350.000000   300.000000   50.000000
(prod5, 1)   800.000522   800.000000    0.000522
(prod5, 2)   399.999338   399.999460    0.000399
(prod5, 3)   599.998374   499.999201   99.999572
(prod5, 4)     0.001027   100.000599    0.000000
(prod5, 5)  1000.006512  1000.000000    0.006512
(prod5, 6)  1149.992981  1099.999493   50.000000
(prod6, 1)   200.000000   200.000000    0.000000
(prod6, 2)   300.000000   300.000000    0.000000
(prod6, 3)   400.000000   400.000000    0.000000
(prod6, 4)     0.000000     0.000000    0.000000
(prod6, 5)   300.000000   300.000000    0.000000
(prod6, 6)   550.000000   500.000000   50.000000
(prod7, 1)   100.000000   100.000000    0.000000
(prod7, 2)   150.000000   150.000000    0.000000
(prod7, 3)   199.999572   100.000000   99.999572
(prod7, 4)     0.000428   100.000000    0.000000
(prod7, 5)     0.000000     0.000000    0.000000
(prod7, 6)   110.000000    60.000000   50.000000
numMachinesDown  (grinder, 1)   -0.000000e+00
numMachinesDown  (grinder, 2)   -0.000000e+00
numMachinesDown  (grinder, 3)   -0.000000e+00
numMachinesDown  (grinder, 4)    2.000000e+00
numMachinesDown  (grinder, 5)   -0.000000e+00
numMachinesDown  (grinder, 6)   -0.000000e+00
numMachinesDown  (vdrill, 1)     0.000000e+00
numMachinesDown  (vdrill, 2)    -0.000000e+00
numMachinesDown  (vdrill, 3)    -0.000000e+00
numMachinesDown  (vdrill, 4)     1.999994e+00
numMachinesDown  (vdrill, 5)     3.955573e-06
numMachinesDown  (vdrill, 6)     2.033239e-06
numMachinesDown  (hdrill, 1)     1.000000e+00
numMachinesDown  (hdrill, 2)     2.000000e+00
numMachinesDown  (hdrill, 3)    -0.000000e+00
numMachinesDown  (hdrill, 4)    -0.000000e+00
```

(continues on next page)

(continued from previous page)

```
numMachinesDown  (hdrill, 5)    -0.000000e+00
numMachinesDown  (hdrill, 6)    -0.000000e+00
numMachinesDown  (borer, 1)      0.000000e+00
numMachinesDown  (borer, 2)     -0.000000e+00
numMachinesDown  (borer, 3)      1.996271e-06
numMachinesDown  (borer, 4)      9.999940e-01
numMachinesDown  (borer, 5)     -0.000000e+00
numMachinesDown  (borer, 6)      3.992541e-06
numMachinesDown  (planer, 1)     0.000000e+00
numMachinesDown  (planer, 2)     1.798545e-06
numMachinesDown  (planer, 3)     1.996271e-06
numMachinesDown  (planer, 4)     9.999957e-01
numMachinesDown  (planer, 5)     8.120488e-16
numMachinesDown  (planer, 6)     4.829074e-07
dtype: float64
Selected Rows from Table SOLUTIONSUMMARY

                                              Value
Label
Solver                                         MILP
Algorithm                            Branch and Cut
Objective Function                     total_profit
Solution Status        Optimal within Relative Gap
Objective Value                     108855.00961

Relative Gap                           3.0634424E-6
Absolute Gap                           0.3334720743
Primal Infeasibility                   5.684342E-14
Bound Infeasibility                               0
Integer Infeasibility                  5.9888119E-6

Best Bound                             108855.34308
Nodes                                             1
Solutions Found                                   3
Iterations                                      377
Presolve Time                                  0.00
Solution Time                                  0.34
Selected Rows from Table PROBLEMSUMMARY

                                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
Fixed                              13
Binary                              0
Integer                            30

Number of Constraints              77
Linear LE (<=)                     30
Linear EQ (=)                      47
```

(continues on next page)

(continued from previous page)

```
Linear GE (>=)                  0
Linear Range                    0

Constraint Coefficients       341
Out[8]: 108855.00960810453
```

4.1.5 Manpower Planning

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex5_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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'],
        ['skilled', 'unskilled']
        ], columns=['worker1', 'worker2'])

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

(continues on next page)

(continued from previous page)

```
# 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')
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)
```

(continues on next page)

(continued from previous page)

```
                           - 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.expr_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.expr_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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.manpower_planning import test

In [8]: test(cas_conn)
NOTE: Initialized model manpower_planning.
NOTE: Added action set 'optimization'.
NOTE: Converting model manpower_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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 0.01 seconds.
NOTE: The LP presolver removed 36 variables and 12 constraints.
NOTE: The LP presolver removed 52 constraint coefficients.
NOTE: The presolved problem has 27 variables, 12 constraints, and 56 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         17    8.417969E+02         0
NOTE: Optimal.
NOTE: Objective = 841.796875.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 24 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
Redundancy: 841.796875
Cost: 1668750.0
               numWorkers  numRecruits  numRedundant  numShortTime  numExcess
unskilled   0  2000.00000          NaN           NaN           NaN        NaN
unskilled   1  1157.03125          0.0    442.968750          50.0  132.03125
unskilled   2   675.00000          0.0    166.328125          50.0  150.00000
unskilled   3   175.00000          0.0    232.500000          50.0  150.00000
semiskilled 0  1500.00000          NaN           NaN           NaN        NaN
semiskilled 1  1442.96875          0.0      0.000000          50.0   17.96875
semiskilled 2  2025.00000        800.0      0.000000          50.0    0.00000
semiskilled 3  2500.00000        800.0      0.000000           0.0    0.00000
skilled     0  1000.00000          NaN           NaN           NaN        NaN
skilled     1  1025.00000          0.0      0.000000          50.0    0.00000
skilled     2  1500.00000        500.0      0.000000           0.0    0.00000
skilled     3  2000.00000        500.0      0.000000           0.0    0.00000
                              numRetrain
(unskilled, semiskilled, 1)  200.000000
(unskilled, semiskilled, 2)  200.000000
(unskilled, semiskilled, 3)  200.000000
(semiskilled, skilled, 1)    256.250000
(semiskilled, skilled, 2)    262.276786
(semiskilled, skilled, 3)    364.285714
```

(continues on next page)

(continued from previous page)

```
                             numDowngrade
(semiskilled, unskilled, 1)  0.000000e+00
(semiskilled, unskilled, 2) -1.421085e-14
(semiskilled, unskilled, 3)  0.000000e+00
(skilled, semiskilled, 1)    1.684375e+02
(skilled, semiskilled, 2)    1.729129e+02
(skilled, semiskilled, 3)    2.210714e+02
(skilled, unskilled, 1)      0.000000e+00
(skilled, unskilled, 2)      0.000000e+00
(skilled, unskilled, 3)      0.000000e+00
NOTE: Added action set 'optimization'.
NOTE: Converting model manpower_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 63 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 24 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
Redundancy: 1423.7188365650968
Cost: 498677.2853185596
              numWorkers  numRecruits  numRedundant  numShortTime  numExcess
unskilled   0     2000.0          NaN           NaN           NaN        NaN
unskilled   1     1000.0     0.000000    812.500000           0.0        0.0
unskilled   2      500.0     0.000000    257.617729           0.0        0.0
unskilled   3        0.0     0.000000    353.601108           0.0        0.0
semiskilled 0     1500.0          NaN           NaN           NaN        NaN
semiskilled 1     1400.0     0.000000      0.000000           0.0        0.0
semiskilled 2     2000.0   800.000000      0.000000           0.0        0.0
semiskilled 3     2500.0   800.000000      0.000000           0.0        0.0
skilled     0     1000.0          NaN           NaN           NaN        NaN
skilled     1     1000.0    55.555556      0.000000           0.0        0.0
skilled     2     1500.0   500.000000      0.000000           0.0        0.0
skilled     3     2000.0   500.000000      0.000000           0.0        0.0
                            numRetrain
```

(continues on next page)

(continued from previous page)

```
(unskilled, semiskilled, 1)    0.000000
(unskilled, semiskilled, 2)  142.382271
(unskilled, semiskilled, 3)   96.398892
(semiskilled, skilled, 1)      0.000000
(semiskilled, skilled, 2)    105.263158
(semiskilled, skilled, 3)    131.578947
                             numDowngrade
(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[8]: 498677.2853185596
```

4.1.6 Refinery Optimization

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex6_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/mpex06.html

Model

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


def test(cas_conn, **kwargs):

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

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

    arc_data = pd.DataFrame([
        ['source', 'crude1', 6],
        ['source', 'crude2', 6],
        ['crude1', 'light_naphtha', 0.1],
        ['crude1', 'medium_naphtha', 0.2],
        ['crude1', 'heavy_naphtha', 0.2],
        ['crude1', 'light_oil', 0.12],
        ['crude1', 'heavy_oil', 0.2],
        ['crude1', 'residuum', 0.13],
        ['crude2', 'light_naphtha', 0.15],
```

(continues on next page)

(continued from previous page)

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

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

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

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

    fuel_oil_ratio_data = pd.DataFrame([
        ['light_oil', 10],
```

(continues on next page)

(continued from previous page)

```
        ['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.expr_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.expr_sum(flow[a] for a in ARCS if a[0] == n) ==
                      so.expr_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')

    octane = octane_data['octane']
    PETROLS = petrol_data.index.tolist()
```

(continues on next page)

(continued from previous page)

```
    octane_lb = petrol_data['octane_lb']
    vapour_pressure = vapour_pressure_data['vapour_pressure']

    m.add_constraints((so.expr_sum(octane[a[0]] * arc_mult[a] * flow[a]
                                    for a in ARCS if a[1] == p)
                       >= octane_lb[p] *
                       so.expr_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.expr_sum(vapour_pressure[a[0]] * arc_mult[a] * flow[a]
                                 for a in ARCS if a[1] == 'jet_fuel') <=
                     vapour_pressure_ub *
                     so.expr_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.expr_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.expr_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(**kwargs)
    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.expr_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))
        octane_sol = pd.Series(octane_sol, name='octane_sol', index=PETROLS)
        print(so.get_solution_table(octane_sol, octane_lb))
```

(continues on next page)

(continued from previous page)

```
        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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.refinery_optimization import test

In [8]: 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 8 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.
```

(continues on next page)

(continued from previous page)

```
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 0.02 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 51 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 46 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
        crudesDistilled
crude1          15000.0
crude2          30000.0
                  oilCracked
light_oil_cracked      4200.0
heavy_oil_cracked      3800.0
                                                 flow
(source, crude1)                         15000.000000
(source, crude2)                         30000.000000
(crude1, light_naphtha)                  15000.000000
(crude1, medium_naphtha)                 15000.000000
(crude1, heavy_naphtha)                  15000.000000
(crude1, light_oil)                      15000.000000
(crude1, heavy_oil)                      15000.000000
(crude1, residuum)                       15000.000000
(crude2, light_naphtha)                  30000.000000
(crude2, medium_naphtha)                 30000.000000
(crude2, heavy_naphtha)                  30000.000000
(crude2, light_oil)                      30000.000000
(crude2, heavy_oil)                      30000.000000
(crude2, residuum)                       30000.000000
(light_naphtha, regular_petrol)           3293.112993
(light_naphtha, premium_petrol)           2706.887007
(medium_naphtha, regular_petrol)         10500.000000
(medium_naphtha, premium_petrol)             0.000000
(heavy_naphtha, regular_petrol)           1315.334140
(heavy_naphtha, premium_petrol)           1677.804016
(light_naphtha, reformed_gasoline)           0.000000
(medium_naphtha, reformed_gasoline)          0.000000
(heavy_naphtha, reformed_gasoline)        5406.861844
(light_oil, jet_fuel)                        0.000000
(light_oil, fuel_oil)                        0.000000
(heavy_oil, jet_fuel)                     4900.000000
(heavy_oil, fuel_oil)                        0.000000
(light_oil, light_oil_cracked)            4200.000000
(light_oil_cracked, cracked_oil)          4200.000000
(light_oil_cracked, cracked_gasoline)     4200.000000
(heavy_oil, heavy_oil_cracked)            3800.000000
(heavy_oil_cracked, cracked_oil)          3800.000000
```

(continues on next page)

(continued from previous page)

```
(heavy_oil_cracked, cracked_gasoline)   3800.000000
(cracked_oil, jet_fuel)                 5706.000000
(cracked_oil, fuel_oil)                    0.000000
(reformed_gasoline, regular_petrol)        0.000000
(reformed_gasoline, premium_petrol)     2433.087830
(cracked_gasoline, regular_petrol)      1936.000000
(cracked_gasoline, premium_petrol)         0.000000
(residuum, lube_oil)                    1000.000000
(residuum, jet_fuel)                    4550.000000
(residuum, fuel_oil)                       0.000000
(premium_petrol, sink)                  6817.778853
(regular_petrol, sink)                 17044.447133
(jet_fuel, sink)                       15156.000000
(fuel_oil, sink)                           0.000000
(lube_oil, sink)                         500.000000
                octane_sol octane_lb
petrol
regular_petrol        84.0        84
premium_petrol        94.0        94
            vapour_pressure
oil
light_oil                 1.0
heavy_oil                 0.6
cracked_oil               1.5
residuum                 0.05
Vapour_pressure_sol: 0.7737
            coefficient num_fuel_oil_ratio_sol
light_oil            10                    nan
cracked_oil           4                    nan
heavy_oil             3                    nan
residuum              1                    nan
Out[8]: 211365.13476893297
```

4.1.7 Mining Optimization

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex7_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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([
```

(continues on next page)

(continued from previous page)

```
        ['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'])

    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.expr_sum(discount[j] * extractedPerYear[j] for j in YEARS)
    totalCost = so.expr_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.expr_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()
```

(continues on next page)

(continued from previous page)

```
    if res is not None:
        print(so.get_solution_table(isOpen, isWorked, extract))
        quality_sol = {j: so.expr_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'])
        print(so.get_solution_table(epy, qs, quality_required))

    return m.get_objective_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.mining_optimization import test

In [8]: 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 8 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 8 variables and 8 constraints.
NOTE: The MILP presolver removed 16 constraint coefficients.
NOTE: The MILP presolver modified 8 constraint coefficients.
NOTE: The presolved problem has 52 variables, 58 constraints, and 135 constraint␣
↪coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 8 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1     14     96.5802313    157.7309278   38.77%       0
                0        1     14     96.5802313    150.9548680   36.02%       0
                0        1     14     96.5802313    147.3693449   34.46%       0
                0        1     15    146.8619974    146.8619974    0.00%       0
```

(continues on next page)

(continued from previous page)

```
                0         0      15     146.8619974    146.8619974     0.00%        0
NOTE: The MILP solver added 7 cuts with 34 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 146.86199735.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 60 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 66 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
              isOpen  isWorked       extract
(mine1, 1)  1.000000  1.000000  2.000000e+00
(mine1, 2)  0.999994  0.000006  1.252000e-05
(mine1, 3)  0.999994  0.999994  1.950000e+00
(mine1, 4)  0.999994  0.999994  1.250007e-01
(mine1, 5)  0.999994  0.999994  1.999987e+00
(mine2, 1)  1.000000  0.000000  0.000000e+00
(mine2, 2)  1.000000  0.999995  2.499988e+00
(mine2, 3)  0.999998 -0.000000  0.000000e+00
(mine2, 4)  0.999998  0.999998  2.499994e+00
(mine2, 5)  0.999998  0.999998  2.166667e+00
(mine3, 1)  1.000000  1.000000  1.300000e+00
(mine3, 2)  1.000000  0.999998  1.299997e+00
(mine3, 3)  1.000000  1.000000  1.300000e+00
(mine3, 4)  1.000000  0.000001  3.477765e-07
(mine3, 5)  1.000000  1.000000  1.300000e+00
(mine4, 1)  1.000000  1.000000  2.450000e+00
(mine4, 2)  1.000000  1.000000  2.200005e+00
(mine4, 3)  1.000000 -0.000000  0.000000e+00
(mine4, 4)  1.000000  1.000000  3.000000e+00
(mine4, 5) -0.000000 -0.000000  0.000000e+00
   extracted_per_year  quality_sol quality_required
1            5.750000          0.9              0.9
2            6.000003          0.8              0.8
3            3.250000          1.2              1.2
4            5.624995          0.6              0.6
5            5.466654          1.0              1.0
Out[8]: 146.861997350257
```

4.1.8 Farm Planning

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex8_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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,
                   '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
```

(continues on next page)

(continued from previous page)

```
    grain_other_costs = 15
    sugar_beet_cost = 70
    sugar_beet_revenue = 58
    sugar_beet_labour_req = 14
    sugar_beet_other_costs = 10
    nominal_labour_cost = 4000
    nominal_labour_hours = 5500
    excess_labour_cost = 1.2
    capital_outlay_unit = 200
    num_loan_years = 10
    annual_interest_rate = 0.15
    max_decrease_ratio = 0.50
    max_increase_ratio = 0.75

    # Sets

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

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

    # Variables

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

(continues on next page)

(continued from previous page)

```
    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.expr_sum(milk_revenue[age] * numCows[age, year]
                           for age in AGES) +
               grain_revenue * grainSold[year] +
               sugar_beet_revenue * sugarBeetSold[year]
               for year in YEARS}

    cost = {year:
            grain_cost * grainBought[year] +
            sugar_beet_cost * sugarBeetBought[year] +
            nominal_labour_cost +
            excess_labour_cost * numExcessLabourHours[year] +
            so.expr_sum(cow_other_costs[age] * numCows[age, year]
                        for age in AGES) +
            so.expr_sum(grain_other_costs * grainAcres[group, year]
                        for group in GROUPS) +
            sugar_beet_other_costs * sugarBeetAcres[year] +
            so.expr_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.expr_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.expr_sum(acres_needed[age] * numCows[age, year] for age in AGES) +
        so.expr_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.expr_sum(
            bullock_yield[age] * numCows[age, year] for age in AGES)
```

(continues on next page)

(continued from previous page)

```
        for year in YEARS), name='numBullocksSold_def')

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

    m.add_constraints((
        so.expr_sum(numCows[age, year] for age in AGES) <= max_num_cows +
        so.expr_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.expr_sum(grain_req[age] * numCows[age, year] for age in AGES) <=
        so.expr_sum(grainGrown[group, year] for group in GROUPS)
        + grainBought[year] - grainSold[year]
        for year in YEARS), name='grain_req_def')

    sugarBeetGrown = {(year): sugar_beet_yield * sugarBeetAcres[year]
                      for year in YEARS}
    m.add_constraints((
        so.expr_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.expr_sum(cow_labour_req[age] * numCows[age, year]
                    for age in AGES) +
        so.expr_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.expr_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:
        so.pd.display_all()
        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'])
```

(continues on next page)

(continued from previous page)

```
        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 = m.get_constraint('num_acres')
        na_df = num_acres.get_expressions()
        max_num_cows_con = m.get_constraint('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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.farm_planning import test

In [8]: test(cas_conn)
NOTE: Initialized model farm_planning.
NOTE: Added action set 'optimization'.
NOTE: Converting model farm_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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 0.01 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.
```

(continues on next page)

(continued from previous page)

```
NOTE: Objective = 121719.17286.
NOTE: The Dual Simplex solve time is 0.02 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 143 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 101 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
          numCows
(0, 0)   10.000000
(0, 1)   22.800000
(0, 2)   11.584427
(0, 3)    0.000000
(0, 4)    0.000000
(0, 5)    0.000000
(1, 0)   10.000000
(1, 1)    9.500000
(1, 2)   21.660000
(1, 3)   11.005205
(1, 4)    0.000000
(1, 5)    0.000000
(2, 0)   10.000000
(2, 1)    9.500000
(2, 2)    9.025000
(2, 3)   20.577000
(2, 4)   10.454945
(2, 5)    0.000000
(3, 0)   10.000000
(3, 1)    9.800000
(3, 2)    9.310000
(3, 3)    8.844500
(3, 4)   20.165460
(3, 5)   10.245846
(4, 0)   10.000000
(4, 1)    9.800000
(4, 2)    9.604000
(4, 3)    9.123800
(4, 4)    8.667610
(4, 5)   19.762151
(5, 0)   10.000000
(5, 1)    9.800000
(5, 2)    9.604000
(5, 3)    9.411920
(5, 4)    8.941324
(5, 5)    8.494258
(6, 0)   10.000000
(6, 1)    9.800000
(6, 2)    9.604000
(6, 3)    9.411920
(6, 4)    9.223682
(6, 5)    8.762498
(7, 0)   10.000000
(7, 1)    9.800000
(7, 2)    9.604000
(7, 3)    9.411920
(7, 4)    9.223682
```

(continues on next page)

(continued from previous page)

```
(7, 5)    9.039208
(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
   numBullocksSold  numHeifersSold  capitalOutlay  numExcessLabourHours  \
1        53.735000       30.935000            0.0                   0.0
2        52.341850       40.757423            0.0                   0.0
3        57.435807       57.435807            0.0                   0.0
4        56.964286       56.964286            0.0                   0.0
5        50.853436       50.853436            0.0                   0.0

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

(continues on next page)

(continued from previous page)

```
(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  \
1    36.620000        0.0       60.766667       91.150000              0.0
2    35.100200        0.0       62.670049       94.005073              0.0
3    37.836507        0.0       65.100304       97.650456              0.0
4    40.142857        0.0       76.428571      114.642857              0.0
5    33.476475        0.0       87.539208      131.308812              0.0

   sugarBeetSold
1      22.760000
2      27.388173
3      24.550338
4      42.142857
5      66.586258
   num_acres  max_num_cows_def
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[8]: 121719.17286133638
```

4.1.9 Economic Planning

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex9_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/mpex09.html

Model

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


def test(cas_conn):

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

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

(continues on next page)

(continued from previous page)

```
                    '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 = industry_data['init_stocks']
    init_productive_capacity = industry_data['init_productive_capacity']
    demand = industry_data['demand']

    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.expr_sum(
                           production_coeff[i, j] * static_production[j]
                           for j in INDUSTRIES) for i in INDUSTRIES),
                      name='static_con')

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

    final_demand = so.get_value_table(
        static_production)['static_production']

    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(2, num_years+3), lb=0,
                                     name='extra_capacity')

    productive_capacity = so.ImplicitVar(
        (init_productive_capacity[i] +
         so.expr_sum(extra_capacity[i, y] for y in range(2, year+1))
```

(continues on next page)

(continued from previous page)

```
         for i in INDUSTRIES for year in range(1, num_years+2)),
        name='productive_capacity'
    )

    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.expr_sum(production[i, year] for i in INDUSTRIES
                                    for year in [4, 5])
    total_manpower = so.expr_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.expr_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.expr_sum(production_coeff['manpower', j] * production[j, year] +
                     productive_capacity_coeff['manpower', j] *
                     extra_capacity[j, year+1]
                     for j in INDUSTRIES)
        <= manpower_capacity for year in range(1, num_years+2)),
        name='manpower_con')

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

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

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

    problem1 = so.Model(name='Problem1', session=cas_conn)
    problem1.include(
        production, stock, extra_capacity, continuity_con, manpower_con,
        capacity_con, productive_capacity)
    problem1.set_objective(total_productive_capacity, sense=so.MAX,
                           name='total_productive_capacity')
    problem1.solve()
    so.pd.display_dense()
    print(so.get_value_table(production, stock, extra_capacity,
```

(continues on next page)

(continued from previous page)

```
                                productive_capacity).sort_index())
    print(so.get_value_table(manpower_con.get_expressions()))

    # Problem 2

    problem2 = so.Model(name='Problem2', session=cas_conn)
    problem2.include(problem1)
    problem2.set_objective(total_production, name='total_production',
                           sense=so.MAX)
    for i in INDUSTRIES:
        for year in YEARS:
            continuity_con[i, year].set_rhs(0)
    problem2.solve()
    print(so.get_value_table(production, stock, extra_capacity,
                                productive_capacity).sort_index())
    print(so.get_value_table(manpower_con.get_expressions()))

    # Problem 3

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

    return problem3.get_objective_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.economic_planning import test

In [8]: test(cas_conn)
NOTE: Initialized model economic_planning.
NOTE: Added action set 'optimization'.
NOTE: Converting model economic_planning to OPTMODEL.
NOTE: Submitting OPTMODEL code to CAS server.
```

(continues on next page)

(continued from previous page)

```
NOTE: Problem generation will use 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 3 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 3 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
           static_production
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 8 threads.
NOTE: The problem has 60 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 15 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 60 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 42 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
             production         stock  extra_capacity  productive_capacity
coal      0    0.000000  1.500000e+02             NaN                  NaN
coal      1  260.402615  0.000000e+00             NaN           300.000000
```

(continues on next page)

(continued from previous page)

```
coal      2  293.406208  0.000000e+00        0.000000           300.000000
coal      3  300.000000  0.000000e+00        0.000000           300.000000
coal      4   17.948718  1.484480e+02      189.203132           489.203132
coal      5  166.396761  0.000000e+00     1022.672065          1511.875197
coal      6  166.396761  1.421085e-14        0.000000          1511.875197
coal      7         NaN           NaN        0.000000                  NaN
steel     0    0.000000  8.000000e+01             NaN                  NaN
steel     1  135.341540  1.228110e+01             NaN           350.000000
steel     2  181.659854  0.000000e+00        0.000000           350.000000
steel     3  193.090418  0.000000e+00        0.000000           350.000000
steel     4  105.668016  0.000000e+00        0.000000           350.000000
steel     5  105.668016  0.000000e+00        0.000000           350.000000
steel     6  105.668016 -1.456613e-13        0.000000           350.000000
steel     7         NaN           NaN        0.000000                  NaN
transport 0    0.000000  1.000000e+02             NaN                  NaN
transport 1  140.722422  6.240839e+00             NaN           280.000000
transport 2  200.580168  0.000000e+00        0.000000           280.000000
transport 3  267.152497  0.000000e+00        0.000000           280.000000
transport 4   92.307692  0.000000e+00        0.000000           280.000000
transport 5   92.307692  0.000000e+00        0.000000           280.000000
transport 6   92.307692 -3.907985e-14        0.000000           280.000000
transport 7         NaN           NaN        0.000000                  NaN
   manpower_con
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 8 threads.
NOTE: The problem has 60 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 15 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.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 60 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 42 rows and 4 columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
             production       stock  extra_capacity  productive_capacity
coal      0    0.000000  150.000000             NaN                  NaN
coal      1  184.818327   31.628509             NaN           300.000000
coal      2  430.504654   16.372454    1.305047e+02           430.504654
coal      3  430.504654    0.000000    0.000000e+00           430.504654
coal      4  430.504654    0.000000   -1.207321e-12           430.504654
coal      5  430.504654    0.000000    0.000000e+00           430.504654
coal      6  166.396761  324.107893    0.000000e+00           430.504654
coal      7         NaN         NaN    0.000000e+00                  NaN
steel     0    0.000000   80.000000             NaN                  NaN
steel     1   86.729504   11.532298             NaN           350.000000
steel     2  155.337478    0.000000    0.000000e+00           350.000000
steel     3  182.867219    0.000000    0.000000e+00           350.000000
steel     4  359.402270    0.000000    9.402270e+00           359.402270
steel     5  359.402270  176.535051    0.000000e+00           359.402270
steel     6  105.668016  490.269305    0.000000e+00           359.402270
steel     7         NaN         NaN    0.000000e+00                  NaN
transport 0    0.000000  100.000000             NaN                  NaN
transport 1  141.312267    0.000000             NaN           280.000000
transport 2  198.387943    0.000000    0.000000e+00           280.000000
transport 3  225.917684    0.000000    0.000000e+00           280.000000
transport 4  519.382633    0.000000    2.393826e+02           519.382633
transport 5  519.382633  293.464949    0.000000e+00           519.382633
transport 6   92.307692  750.539890    0.000000e+00           519.382633
transport 7         NaN         NaN    0.000000e+00                  NaN
   manpower_con
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 8 threads.
NOTE: The problem has 60 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 15 variables and 3 constraints.
NOTE: The LP presolver removed 31 constraint coefficients.
NOTE: The presolved problem has 45 variables, 33 constraints, and 188 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.464016E+05         0
```

(continues on next page)

(continued from previous page)

```
          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 0.01 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 60 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 36 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
             production       stock  extra_capacity  productive_capacity
coal      0    0.000000  150.000000             NaN                  NaN
coal      1  251.792754    0.000000             NaN           300.000000
coal      2  316.015222    0.000000       16.015222           316.015222
coal      3  319.832020    0.000000        3.816798           319.832020
coal      4  366.349753    0.000000       46.517734           366.349753
coal      5  859.359606    0.000000      493.009853           859.359606
coal      6  859.359606  460.207993        0.000000           859.359606
coal      7         NaN         NaN        0.000000                  NaN
steel     0    0.000000   80.000000             NaN                  NaN
steel     1  134.794583   11.028028             NaN           350.000000
steel     2  175.041379    0.000000        0.000000           350.000000
steel     3  224.064039    0.000000        0.000000           350.000000
steel     4  223.136289    0.000000        0.000000           350.000000
steel     5  220.043787    0.000000        0.000000           350.000000
steel     6  350.000000    0.000000        0.000000           350.000000
steel     7         NaN         NaN        0.000000                  NaN
transport 0    0.000000  100.000000             NaN                  NaN
transport 1  143.558583    4.247230             NaN           280.000000
transport 2  181.676355    0.000000        0.000000           280.000000
transport 3  280.000000    0.000000        0.000000           280.000000
transport 4  279.072249    0.000000        0.000000           280.000000
transport 5  275.979748    0.000000        0.000000           280.000000
transport 6  195.539132    0.000000        0.000000           280.000000
transport 7         NaN         NaN        0.000000                  NaN
   manpower_con
1    226.631832
2    279.983537
3    333.725517
4    539.769130
5    636.824849
6    659.723590
Out[8]: 2450.026622821294
```

4.1.10 Decentralization

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex10_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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.expr_sum(benefit[i, j] * assign[i, j]
                           for i in DEPTS for j in CITIES)

totalCost = so.expr_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.expr_sum(assign[dept, city] for city in CITIES)
                  == 1 for dept in DEPTS), name='assign_dept')

m.add_constraints((so.expr_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.expr_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.expr_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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.decentralization import test

In [8]: 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 8 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.
```

(continues on next page)

(continued from previous page)

```
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 8 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%       0
                0        1      3      8.1000000     33.3692308   75.73%       0
                0        1      3      8.1000000     32.6500000   75.19%       0
                0        1      3      8.1000000     31.9066667   74.61%       0
                0        1      3      8.1000000     30.7000000   73.62%       0
                0        1      3      8.1000000     30.1600000   73.14%       0
                0        1      3      8.1000000     29.8800000   72.89%       0
                0        1      3      8.1000000     29.8000000   72.82%       0
                0        1      3      8.1000000     29.4722222   72.52%       0
                0        1      3      8.1000000     28.9117647   71.98%       0
                0        1      3      8.1000000     28.6716667   71.75%       0
                0        1      3      8.1000000     28.5000000   71.58%       0
                0        1      4     14.9000000     14.9000000    0.00%       0
NOTE: The MILP solver added 34 cuts with 185 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 14.9.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 105 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 278 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
Selected Rows from Table PROBLEMSUMMARY

                                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
```

(continues on next page)

(continued from previous page)

```
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 8 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.
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 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 105 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 68 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
Selected Rows from Table PROBLEMSUMMARY

                                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
```

(continues on next page)

(continued from previous page)

```
Linear EQ (=)                        65
Linear GE (>=)                        0
Linear Range                          0

Constraint Coefficients             270
   totalBenefit  totalCost
-           80.0       65.1
assign  (A, Bristol)      1.0
        (A, Brighton)     0.0
        (A, London)       0.0
        (B, Bristol)      0.0
        (B, Brighton)     1.0
        (B, London)       0.0
        (C, Bristol)      0.0
        (C, Brighton)     1.0
        (C, London)      -0.0
        (D, Bristol)      1.0
        (D, Brighton)     0.0
        (D, London)       0.0
        (E, Bristol)      0.0
        (E, Brighton)     1.0
        (E, London)       0.0
dtype: float64
Out[8]: 14.9
```

4.1.11 Optimal Wedding Seating

Reference

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

Model

```python
import sasoptpy as so
import math


def test(cas_conn, num_guests=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)
```

(continues on next page)

(continued from previous page)

```
    # 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))

        # 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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.sas_optimal_wedding import test

In [8]: 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 TMP121JB6BH␣
↪in caslib CASUSER(casuser).
NOTE: The table TMP121JB6BH has been created in caslib CASUSER(casuser) from binary␣
↪data uploaded to Cloud Analytic Services.
```

(continues on next page)

(continued from previous page)

```
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 8 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    0    0
            .       0.0000      13.0000     13.0000 1.30e+01 1.30e+01    1    2
           10       0.0000      13.0000     13.0000 1.30e+01 1.30e+01    1    2
           18       4.2500      13.0000     13.0000  205.88%  205.88%    5    7
           19       6.0000      13.0000     13.0000  116.67%  116.67%    6    7
            .       6.0000      13.0000     13.0000  116.67%  116.67%    6    7
           20       6.0000      13.0000     13.0000  116.67%  116.67%    6    8
           21       9.5000      13.0000     13.0000   36.84%   36.84%    7    8
           23      13.0000      13.0000     13.0000    0.00%    0.00%    7    9
            Node  Active   Sols         Best         Best      Gap    CPU   Real
                                     Integer        Bound            Time   Time
               0       1      3      13.0000      13.0000    0.00%      7      9
NOTE: The Decomposition algorithm used 8 threads.
NOTE: The Decomposition algorithm time is 9.30 seconds.
NOTE: Optimal.
NOTE: Objective = 13.
           x
(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
```

(continues on next page)

(continued from previous page)

```
(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
(5, 3)   0.0
(5, 4)   0.0
(5, 5)   0.0
(5, 6)   0.0
(5, 7)   0.0
(6, 1)   0.0
(6, 2)   1.0
(6, 3)   0.0
(6, 4)   0.0
(6, 5)   0.0
(6, 6)   0.0
(6, 7)   0.0
(7, 1)   0.0
(7, 2)   0.0
(7, 3)   1.0
(7, 4)   0.0
(7, 5)   0.0
(7, 6)   0.0
(7, 7)   0.0
(8, 1)   0.0
(8, 2)   0.0
(8, 3)   1.0
(8, 4)   0.0
(8, 5)   0.0
(8, 6)   0.0
(8, 7)   0.0
(9, 1)   0.0
(9, 2)   0.0
(9, 3)   1.0
(9, 4)   0.0
(9, 5)   0.0
(9, 6)   0.0
(9, 7)   0.0
(10, 1)  0.0
(10, 2)  0.0
(10, 3)  0.0
(10, 4)  1.0
(10, 5)  0.0
(10, 6)  0.0
(10, 7)  0.0
(11, 1)  0.0
(11, 2)  0.0
(11, 3)  0.0
(11, 4)  1.0
```

(continues on next page)

(continued from previous page)

```
(11, 5)   0.0
(11, 6)   0.0
(11, 7)   0.0
(12, 1)   0.0
(12, 2)   0.0
(12, 3)   0.0
(12, 4)   1.0
(12, 5)   0.0
(12, 6)   0.0
(12, 7)   0.0
(13, 1)   0.0
(13, 2)   0.0
(13, 3)   0.0
(13, 4)   0.0
(13, 5)   1.0
(13, 6)   0.0
(13, 7)   0.0
(14, 1)   0.0
(14, 2)   0.0
(14, 3)   0.0
(14, 4)   0.0
(14, 5)   1.0
(14, 6)   0.0
(14, 7)   0.0
(15, 1)   0.0
(15, 2)   0.0
(15, 3)   0.0
(15, 4)   0.0
(15, 5)   1.0
(15, 6)   0.0
(15, 7)   0.0
(16, 1)   0.0
(16, 2)   0.0
(16, 3)   0.0
(16, 4)   0.0
(16, 5)   0.0
(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
(19, 1)   0.0
(19, 2)   0.0
(19, 3)   0.0
(19, 4)   0.0
(19, 5)   0.0
```

(continues on next page)

(continued from previous page)

```
(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
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[8]: 13.0
```

4.1.12 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, **kwargs):
    # 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
    model = so.Model("kidney_exchange", session=cas_conn)

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

    # Variables
```

(continues on next page)

(continued from previous page)

```
    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.expr_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}, **kwargs)

    # 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(options={
        'with': 'milp', 'maxtime': 300, 'presolver': 'basic',
        'decomp': {'method': 'user'}}, **kwargs)

    return model.get_objective_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)
```

(continues on next page)

(continued from previous page)

```
In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.sas_kidney_exchange import test

In [8]: 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 8 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 299.83 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 8 threads.
             Node   Active   Sols    BestInteger      BestBound      Gap    Time
                0        1      4     12.0775275   2279.8770216   99.47%       0
                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%       1
                0        1      4     12.0775275     18.3085704   34.03%       1
                0        1      4     12.0775275     18.3085704   34.03%       1
                0        1      4     12.0775275     18.3085704   34.03%       1
NOTE: The MILP solver added 4 cuts with 196 cut coefficients at the root.
               11        8      5     17.1113590     18.1252274    5.59%       1
               28        6      6     17.1113590     18.0210902    5.05%       2
               30        5      7     17.1113590     18.0210902    5.05%       2
               40        6      8     17.1113590     18.0210902    5.05%       2
               66        0      8     17.1113590     17.1113590    0.00%       2
NOTE: Optimal.
NOTE: Objective = 17.111358985.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 8133 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 5967 rows and 4␣
↪columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
NOTE: Added action set 'optimization'.
```

(continues on next page)

(continued from previous page)

```
NOTE: Converting model kidney_exchange to DataFrame.
NOTE: Cloud Analytic Services made the uploaded file available as table BLOCKSTABLE␣
↪in caslib CASUSER(casuser).
NOTE: The table BLOCKSTABLE has been created in caslib CASUSER(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 TMP8_M0VGSY␣
↪in caslib CASUSER(casuser).
NOTE: The table TMP8_M0VGSY has been created in caslib CASUSER(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 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 8 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%    1    1
            1     259.0121      10.5814      10.5814   95.91%    95.91%    1    1
            2     230.6758      10.5814      10.5814   95.41%    95.41%    1    2
            3     204.2627      10.5814      10.5814   94.82%    94.82%    2    2
            4     192.9770      14.7394      14.7394   92.36%    92.36%    2    2
            6     162.6582      15.6274      15.6274   90.39%    90.39%    6    7
            7     140.2584      15.6274      15.6274   88.86%    88.86%    6    7
            8     109.9454      15.6274      15.6274   85.79%    85.79%    7    7
            9      65.4530      16.1007      15.6274   75.40%    76.12%    7    8
           10      65.4530      16.1007      17.1114   75.40%    73.86%    7    8
           11      51.8662      17.1114      17.1114   67.01%    67.01%    7    8
           14      25.6849      17.1114      17.1114   33.38%    33.38%    8    9
           15      17.1114      17.1114      17.1114    0.00%     0.00%    8   10
            Node  Active   Sols         Best         Best      Gap     CPU    Real
                                     Integer        Bound             Time    Time
               0       1      9      17.1114      17.1114    0.00%       8      10
NOTE: The Decomposition algorithm used 8 threads.
NOTE: The Decomposition algorithm time is 10.48 seconds.
NOTE: Optimal.
```

(continues on next page)

(continued from previous page)

```
NOTE: Objective = 17.111358985.
Out[8]: 17.11135898487
```

4.1.13 Multiobjective

Reference

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

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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)
    f2 = m.append_objective((x[1]-x[2])**2 + (x[2] - 3)**2, name='f2', sense=so.MIN)

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

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

    if sols:
        return dict(solutions=cas_conn.CASTable('allsols').to_frame(),
                    x=x, f1=f1, f2=f2)
    else:
        return f1.get_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

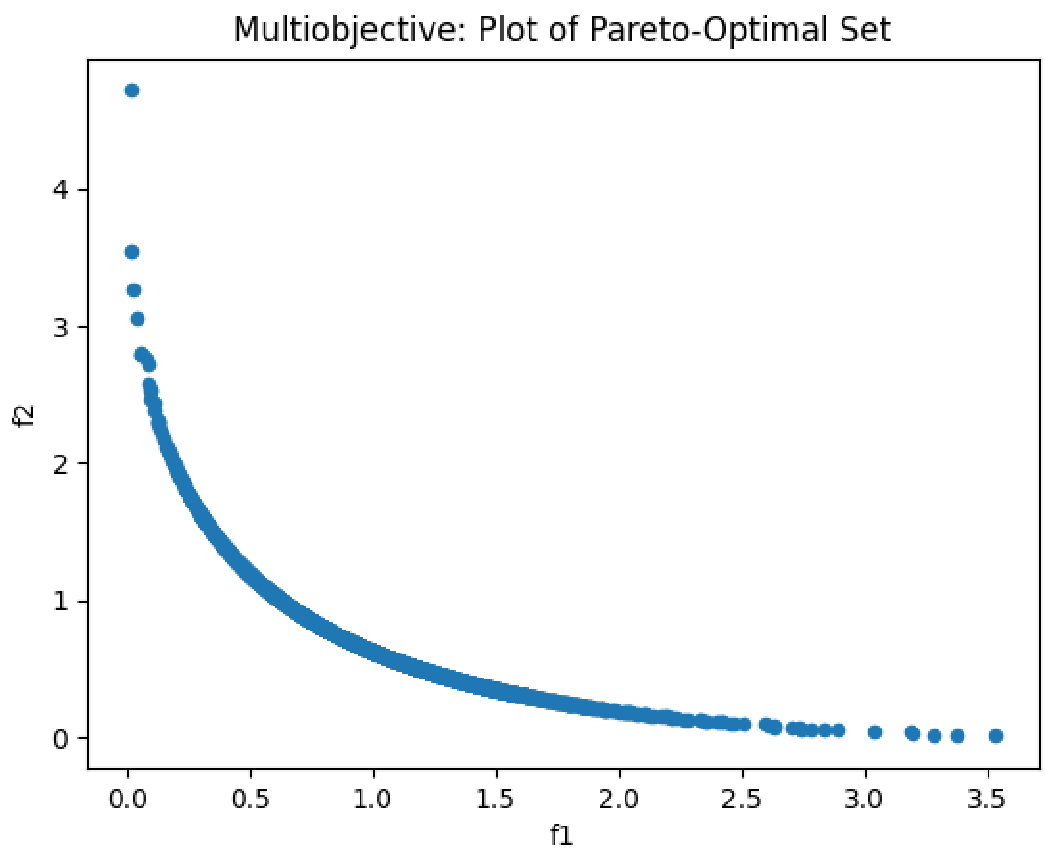
```
In [7]: from examples.client_side.multiobjective import test

In [8]: response = 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 blackbox obj (f1 f2) / logfreq=50;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.lb␣
↪ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_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 8 threads.
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 black-box solver is using up to 8 threads.
NOTE: The black-box 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         0
                 101       1654        0.0000123                0      5576         1
                 151       2290   0.000003230545                0      8181         2
                 201       2874        0.0000182                0     10796         3
                 251       3422   0.000003148003                0     13447         4
                 301       3847   0.000001559509                0     16046         5
                 351       4282   0.000001159917                0     18733         7
                 401       4712   0.000002423148                0     21315         8
                 428       4932   0.000000704951                0     22734        10
NOTE: Function convergence criteria reached.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 2 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 0 rows and 4 columns.
NOTE: The output table 'ALLSOLS' in caslib 'CASUSER(casuser)' has 2 rows and 4934␣
↪columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 14 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
f1 0.014970940561294 93
f2 4.721632618656377
```

```
In [9]: import matplotlib.pyplot as plt

In [10]: sols = response['solutions']
```

(continues on next page)

(continued from previous page)

```
In [11]: x = response['x']

In [12]: f1 = response['f1']

In [13]: f2 = response['f2']

In [14]: tr = sols.transpose()

In [15]: scvalues = tr.iloc[2:]

In [16]: scvalues = scvalues.astype({0: float, 1: float})

In [17]: x[1].set_value(scvalues[0])

In [18]: x[2].set_value(scvalues[1])

In [19]: scvalues['f1'] = f1.get_value()

In [20]: scvalues['f2'] = f2.get_value()

In [21]: f = scvalues.plot.scatter(x='f1', y='f2')

In [22]: f.set_title('Multiobjective: Plot of Pareto-Optimal Set');

In [23]: f
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3e18467128>
```



4.1.14 Least Squares

Reference

SAS/OR example: https://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_nlpsolver_gettingstarted05.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: https://support.sas.com/documentation/onlinedoc/or/ex_code/151/nlpsg01.html

Model

```
import sasoptpy as so
import pandas as pd


def test(cas_conn, data=None):

    # Use default data if not passed
    if data is None:
        data = pd.DataFrame([
            [4, 8, 43.71],
            [62, 5, 351.29],
```

(continues on next page)

(continued from previous page)

```
            [81, 62, 2878.91],
            [85, 75, 3591.59],
            [65, 54, 2058.71],
            [96, 84, 4487.87],
            [98, 29, 1773.52],
            [36, 33, 767.57],
            [30, 91, 1637.66],
            [3, 59, 215.28],
            [62, 57, 2067.42],
            [11, 48, 394.11],
            [66, 21, 932.84],
            [68, 24, 1069.21],
            [95, 30, 1770.78],
            [34, 14, 368.51],
            [86, 81, 3902.27],
            [37, 49, 1115.67],
            [46, 80, 2136.92],
            [87, 72, 3537.84],
        ], columns=['x1', 'x2', 'y'])

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

    # Regression model: L(a,b,c) = a * x1 + b * x2 + c * x1 * x2
    a = m.add_variable(name='a')
    b = m.add_variable(name='b')
    c = m.add_variable(name='c')

    x1 = data['x1']
    x2 = data['x2']
    y = data['y']

    err = m.add_implicit_variable((
        y[i] - (a * x1[i] + b * x2[i] + c * x1[i]  * x2[i]) for i in data.index
    ), name='error')
    m.set_objective(so.expr_sum(err[i]**2 for i in data.index),
                    sense=so.MIN,
                    name='total_error')
    m.solve(verbose=True, options={'with': 'nlp'})
    return m.get_objective_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.least_squares import test

In [8]: test(cas_conn)
NOTE: Initialized model least_squares.
NOTE: Added action set 'optimization'.
NOTE: Converting model least_squares to OPTMODEL.
   var a;
   var b;
   var c;
   impvar error_0 = - 4 * a - 8 * b - 32 * c + 43.71;
   impvar error_1 = - 62 * a - 5 * b - 310 * c + 351.29;
   impvar error_2 = - 81 * a - 62 * b - 5022 * c + 2878.91;
   impvar error_3 = - 85 * a - 75 * b - 6375 * c + 3591.59;
   impvar error_4 = - 65 * a - 54 * b - 3510 * c + 2058.71;
   impvar error_5 = - 96 * a - 84 * b - 8064 * c + 4487.87;
   impvar error_6 = - 98 * a - 29 * b - 2842 * c + 1773.52;
   impvar error_7 = - 36 * a - 33 * b - 1188 * c + 767.57;
   impvar error_8 = - 30 * a - 91 * b - 2730 * c + 1637.66;
   impvar error_9 = - 3 * a - 59 * b - 177 * c + 215.28;
   impvar error_10 = - 62 * a - 57 * b - 3534 * c + 2067.42;
   impvar error_11 = - 11 * a - 48 * b - 528 * c + 394.11;
   impvar error_12 = - 66 * a - 21 * b - 1386 * c + 932.84;
   impvar error_13 = - 68 * a - 24 * b - 1632 * c + 1069.21;
   impvar error_14 = - 95 * a - 30 * b - 2850 * c + 1770.78;
   impvar error_15 = - 34 * a - 14 * b - 476 * c + 368.51;
   impvar error_16 = - 86 * a - 81 * b - 6966 * c + 3902.27;
   impvar error_17 = - 37 * a - 49 * b - 1813 * c + 1115.67;
   impvar error_18 = - 46 * a - 80 * b - 3680 * c + 2136.92;
   impvar error_19 = - 87 * a - 72 * b - 6264 * c + 3537.84;
   min total_error = (- 4 * a - 8 * b - 32 * c + 43.71) ^ (2) + (- 62 * a - 5 * b -␣
↪310 * c + 351.29) ^ (2) + (- 81 * a - 62 * b - 5022 * c + 2878.91) ^ (2) + (- 85 *␣
↪a - 75 * b - 6375 * c + 3591.59) ^ (2) + (- 65 * a - 54 * b - 3510 * c + 2058.71) ^␣
↪(2) + (- 96 * a - 84 * b - 8064 * c + 4487.87) ^ (2) + (- 98 * a - 29 * b - 2842 *␣
↪c + 1773.52) ^ (2) + (- 36 * a - 33 * b - 1188 * c + 767.57) ^ (2) + (- 30 * a - 91␣
↪* b - 2730 * c + 1637.66) ^ (2) + (- 3 * a - 59 * b - 177 * c + 215.28) ^ (2) + (-␣
↪62 * a - 57 * b - 3534 * c + 2067.42) ^ (2) + (- 11 * a - 48 * b - 528 * c + 394.
↪11) ^ (2) + (- 66 * a - 21 * b - 1386 * c + 932.84) ^ (2) + (- 68 * a - 24 * b -␣
↪1632 * c + 1069.21) ^ (2) + (- 95 * a - 30 * b - 2850 * c + 1770.78) ^ (2) + (- 34␣
↪* a - 14 * b - 476 * c + 368.51) ^ (2) + (- 86 * a - 81 * b - 6966 * c + 3902.27) ^␣
↪(2) + (- 37 * a - 49 * b - 1813 * c + 1115.67) ^ (2) + (- 46 * a - 80 * b - 3680 *␣
↪c + 2136.92) ^ (2) + (- 87 * a - 72 * b - 6264 * c + 3537.84) ^ (2);
   solve with nlp / ;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.lb␣
↪ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_CON_.
↪dual;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 threads.
NOTE: The problem has 3 variables (3 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: The NLP solver is called.
NOTE: The Interior Point Direct algorithm is used.
```

(continues on next page)

(continued from previous page)

```
                          Objective                                 Optimality
             Iter             Value      Infeasibility                   Error
                0          95424613                  0              3.48646173
                1        7.18629678                  0    0.0000000055789
NOTE: Optimal.
NOTE: Objective = 7.1862967833.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 3 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 0 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
Out[8]: 7.186296783293
```

Workspace Examples

4.1.15 Efficiency Analysis

Reference

SAS/OR example: https://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex22_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/mpex22.html

Model

```python
import sasoptpy as so
import pandas as pd
from sasoptpy.util import iterate, concat
from sasoptpy.actions import (
    read_data, create_data, cofor_loop, for_loop, solve, if_condition, diff,
    print_item, inline_condition)


def test(cas_conn, get_tables=False):

    input_list = pd.DataFrame(
        ['staff', 'showroom', 'pop1', 'pop2', 'alpha_enq', 'beta_enq'],
        columns=['input'])
    input_data = cas_conn.upload_frame(
        data=input_list, casout={'name': 'input_data', 'replace': True})

    output_list = pd.DataFrame(
        ['alpha_sales', 'beta_sales', 'profit'], columns=['output'])
    output_data = cas_conn.upload_frame(
        data=output_list, casout={'name': 'output_data', 'replace': True})

    problem_data = pd.DataFrame([
        ['Winchester', 7, 8, 10, 12, 8.5, 4, 2, 0.6, 1.5],
        ['Andover', 6, 6, 20, 30, 9, 4.5, 2.3, 0.7, 1.6],
        ['Basingstoke', 2, 3, 40, 40, 2, 1.5, 0.8, 0.25, 0.5],
        ['Poole', 14, 9, 20, 25, 10, 6, 2.6, 0.86, 1.9],
```

(continues on next page)

(continued from previous page)

```
        ['Woking', 10, 9, 10, 10, 11, 5, 2.4, 1, 2],
        ['Newbury', 24, 15, 15, 13, 25, 19, 8, 2.6, 4.5],
        ['Portsmouth', 6, 7, 50, 40, 8.5, 3, 2.5, 0.9, 1.6],
        ['Alresford', 8, 7.5, 5, 8, 9, 4, 2.1, 0.85, 2],
        ['Salisbury', 5, 5, 10, 10, 5, 2.5, 2, 0.65, 0.9],
        ['Guildford', 8, 10, 30, 35, 9.5, 4.5, 2.05, 0.75, 1.7],
        ['Alton', 7, 8, 7, 8, 3, 2, 1.9, 0.7, 0.5],
        ['Weybridge', 5, 6.5, 9, 12, 8, 4.5, 1.8, 0.63, 1.4],
        ['Dorchester', 6, 7.5, 10, 10, 7.5, 4, 1.5, 0.45, 1.45],
        ['Bridport', 11, 8, 8, 10, 10, 6, 2.2, 0.65, 2.2],
        ['Weymouth', 4, 5, 10, 10, 7.5, 3.5, 1.8, 0.62, 1.6],
        ['Portland', 3, 3.5, 3, 2, 2, 1.5, 0.9, 0.35, 0.5],
        ['Chichester', 5, 5.5, 8, 10, 7, 3.5, 1.2, 0.45, 1.3],
        ['Petersfield', 21, 12, 6, 8, 15, 8, 6, 0.25, 2.9],
        ['Petworth', 6, 5.5, 2, 2, 8, 5, 1.5, 0.55, 1.55],
        ['Midhurst', 3, 3.6, 3, 3, 2.5, 1.5, 0.8, 0.2, 0.45],
        ['Reading', 30, 29, 120, 80, 35, 20, 7, 2.5, 8],
        ['Southampton', 25, 16, 110, 80, 27, 12, 6.5, 3.5, 5.4],
        ['Bournemouth', 19, 10, 90, 12, 25, 13, 5.5, 3.1, 4.5],
        ['Henley', 7, 6, 5, 7, 8.5, 4.5, 1.2, 0.48, 2],
        ['Maidenhead', 12, 8, 7, 10, 12, 7, 4.5, 2, 2.3],
        ['Fareham', 4, 6, 1, 1, 7.5, 3.5, 1.1, 0.48, 1.7],
        ['Romsey', 2, 2.5, 1, 1, 2.5, 1, 0.4, 0.1, 0.55],
        ['Ringwood', 2, 3.5, 2, 2, 1.9, 1.2, 0.3, 0.09, 0.4],
    ], columns=['garage_name', 'staff', 'showroom', 'pop1', 'pop2', 'alpha_enq',
                'beta_enq', 'alpha_sales', 'beta_sales', 'profit'])
    garage_data = cas_conn.upload_frame(
        data=problem_data, casout={'name': 'garage_data', 'replace': True})

    with so.Workspace(name='efficiency_analysis', session=cas_conn) as w:
        inputs = so.Set(name='INPUTS', settype=so.string)
        read_data(table=input_data, index={'target': inputs, 'key': 'input'})

        outputs = so.Set(name='OUTPUTS', settype=so.string)
        read_data(table=output_data, index={'target': outputs, 'key': 'output'})

        garages = so.Set(name='GARAGES', settype=so.number)
        garage_name = so.ParameterGroup(garages, name='garage_name', ptype=so.string)
        input = so.ParameterGroup(inputs, garages, name='input')
        output = so.ParameterGroup(outputs, garages, name='output')
        r = read_data(table=garage_data, index={'target': garages, 'key': so.N},
                      columns=[garage_name])
        with iterate(inputs, 'i') as i:
            r.append({'index': i, 'target': input[i, so.N], 'column': i})
        with iterate(outputs, 'i') as i:
            r.append({'index': i, 'target': output[i, so.N], 'column': i})

        k = so.Parameter(name='k', ptype=so.number)
        efficiency_number = so.ParameterGroup(garages, name='efficiency_number')
        weight_sol = so.ParameterGroup(garages, garages, name='weight_sol')

        weight = so.VariableGroup(garages, name='Weight', lb=0)
        inefficiency = so.Variable(name='Inefficiency', lb=0)

        obj = so.Objective(inefficiency, name='Objective', sense=so.maximize)

        input_con = so.ConstraintGroup(
```

(continues on next page)

(continued from previous page)

```
            (so.expr_sum(input[i, j] * weight[j] for j in garages) <= input[i, k]
             for i in inputs), name='input_con')
        output_con = so.ConstraintGroup(
            (so.expr_sum(output[i, j] * weight[j] for j in garages) >= output[i, k] *␣
↪inefficiency
             for i in outputs), name='output_con')

        for kk in cofor_loop(garages):
            k.set_value(kk)
            solve()
            efficiency_number[k] = 1 / inefficiency.sol
            for j in for_loop(garages):
                def if_block():
                    weight_sol[k, j] = weight[j].sol
                def else_block():
                    weight_sol[k, j] = None
                if_condition(weight[j].sol > 1e-6, if_block, else_block)

        efficient_garages = so.Set(
            name='EFFICIENT_GARAGES',
            value=[j.sym for j in garages if j.sym.under_condition(efficiency_
↪number[j] >= 1)])
        inefficient_garages = so.Set(value=diff(garages, efficient_garages), name=
↪'INEFFICIENT_GARAGES')

        p1 = print_item(garage_name, efficiency_number)
        ed = create_data(table='efficiency_data', index={'key': ['garage']}, columns=[
            garage_name, efficiency_number
        ])
        with iterate(inefficient_garages, 'inefficient_garage') as i:
            wd = create_data(table='weight_data_dense',
                             index={'key': [i], 'set': [i.get_set()]},
                             columns=[garage_name, efficiency_number])
            with iterate(efficient_garages, 'efficient_garage') as j:
                wd.append({
                    'name': concat('w', j),
                    'expression': weight_sol[i, j],
                    'index': j
                })

        filtered_set = so.InlineSet(
            lambda: ((g1, g2)
                     for g1 in inefficient_garages
                     for g2 in efficient_garages
                     if inline_condition(weight_sol[g1, g2] != None)))
        wds = create_data(table='weight_data_sparse',
                          index={'key': ['i', 'j'], 'set': [filtered_set]},
                          columns=[weight_sol])

    print(w.to_optmodel())
    w.submit()

    print('Print Table:')
    print(p1.get_response())

    print('Efficiency Data:')
    print(ed.get_response())
```

(continues on next page)

(continued from previous page)

```
    print('Weight Data (Dense):')
    print(wd.get_response())

    print('Weight Data (Sparse):')
    print(wds.get_response())

    if get_tables:
        return obj.get_value(), ed.get_response()
    else:
        return obj.get_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.server_side.efficiency_analysis import test

In [8]: test(cas_conn)
NOTE: Cloud Analytic Services made the uploaded file available as table INPUT_DATA in␣
↪caslib CASUSER(casuser).
NOTE: The table INPUT_DATA has been created in caslib CASUSER(casuser) from binary␣
↪data uploaded to Cloud Analytic Services.
NOTE: Cloud Analytic Services made the uploaded file available as table OUTPUT_DATA␣
↪in caslib CASUSER(casuser).
NOTE: The table OUTPUT_DATA has been created in caslib CASUSER(casuser) from binary␣
↪data uploaded to Cloud Analytic Services.
NOTE: Cloud Analytic Services made the uploaded file available as table GARAGE_DATA␣
↪in caslib CASUSER(casuser).
NOTE: The table GARAGE_DATA has been created in caslib CASUSER(casuser) from binary␣
↪data uploaded to Cloud Analytic Services.
proc optmodel;
   set <str> INPUTS;
   read data INPUT_DATA into INPUTS=[input] ;
   set <str> OUTPUTS;
   read data OUTPUT_DATA into OUTPUTS=[output] ;
   set GARAGES;
   str garage_name {GARAGES};
   num input {INPUTS, GARAGES};
   num output {OUTPUTS, GARAGES};
   read data GARAGE_DATA into GARAGES=[_N_] garage_name {i in INPUTS} < input[i, _N_
↪]=col(i) > {i in OUTPUTS} < output[i, _N_]=col(i) >;
   num k;
   num efficiency_number {GARAGES};
```

(continues on next page)

(continued from previous page)

```
   num weight_sol {GARAGES, GARAGES};
   var Weight {{GARAGES}} >= 0;
   var Inefficiency >= 0;
   max Objective = Inefficiency;
   con input_con {o21 in INPUTS} : input[o21, k] - (sum {j in GARAGES} (input[o21, j]␣
↪* Weight[j])) >= 0;
   con output_con {o33 in OUTPUTS} : sum {j in GARAGES} (output[o33, j] * Weight[j]) -
↪ output[o33, k] * Inefficiency >= 0;
   cofor {o46 in GARAGES} do;
      k = o46;
      solve;
      efficiency_number[k] = (1) / (Inefficiency.sol);
      for {o58 in GARAGES} do;
         if Weight[o58].sol > 1e-06 then do;
            weight_sol[k, o58] = Weight[o58].sol;
         end;
         else do;
            weight_sol[k, o58] = .;
         end;
      end;
   end;
   set EFFICIENT_GARAGES = {{o69 in GARAGES: efficiency_number[o69] >= 1}};
   set INEFFICIENT_GARAGES = GARAGES diff EFFICIENT_GARAGES;
   print garage_name efficiency_number;
   create data efficiency_data from [garage] garage_name efficiency_number;
   create data weight_data_dense from [inefficient_garage] = {{INEFFICIENT_GARAGES}}␣
↪garage_name efficiency_number {efficient_garage in EFFICIENT_GARAGES} < col('w' ||␣
↪efficient_garage)=(weight_sol[inefficient_garage, efficient_garage]) >;
   create data weight_data_sparse from [i j] = {{{o83 in INEFFICIENT_GARAGES, o85 in␣
↪EFFICIENT_GARAGES: weight_sol[o83, o85] ne .}}} weight_sol;
quit;
NOTE: Added action set 'optimization'.
NOTE: There were 6 rows read from table 'INPUT_DATA' in caslib 'CASUSER(casuser)'.
NOTE: There were 3 rows read from table 'OUTPUT_DATA' in caslib 'CASUSER(casuser)'.
NOTE: There were 28 rows read from table 'GARAGE_DATA' in caslib 'CASUSER(casuser)'.
NOTE: The COFOR statement is executing in single-machine mode.
NOTE: Problem generation will use 8 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0.01 seconds.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    2.788571E+01         0
          P 2          6    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
```

(continues on next page)

(continued from previous page)

```
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.185408E+01         0
          P 2          6    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 6 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0.01 seconds.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.669507E+01         0
          P 2         15    1.152977E+00         0
NOTE: Optimal.
NOTE: Objective = 1.1529771581.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 5 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0.01 seconds.
NOTE: The LP presolver removed 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
```

(continues on next page)

(continued from previous page)

```
         Phase Iteration        Value         Time
          D 2          1    6.771196E+01         0
          P 2          5    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 8 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.367804E+01         0
          P 2         18    1.191606E+00         0
NOTE: Optimal.
NOTE: Objective = 1.1916056975.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.282477E+01         0
          P 2          7    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
```

(continues on next page)

(continued from previous page)

```
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    2.113425E+02         0
          P 2          7    1.141723E+00         0
NOTE: Optimal.
NOTE: Objective = 1.141723356.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    8.984164E+01         0
          P 2         20    1.190229E+00         0
NOTE: Optimal.
NOTE: Objective = 1.1902294108.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    7.496160E+01         0
          P 2          7    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 GE, 0 range).
```

(continues on next page)

(continued from previous page)

```
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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.137365E+01         0
          P 2          9    1.011903E+00         0
NOTE: Optimal.
NOTE: Objective = 1.0119030842.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    5.430205E+01         0
          P 2         18    1.018276E+00         0
NOTE: Optimal.
NOTE: Objective = 1.0182756046.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.121193E+01         0
          P 2         17    1.170487E+00         0
```

(continues on next page)

(continued from previous page)

```
NOTE: Optimal.
NOTE: Objective = 1.170487106.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    7.130206E+01         0
          P 2         12    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    7.942787E+01         0
          P 2         15    1.090062E+00         0
NOTE: Optimal.
NOTE: Objective = 1.0900621118.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 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    3.333800E+01         0
          P 2          8    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    3.269636E+01         0
          P 2          5    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.483009E+02         0
          P 2         12    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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.
```

(continues on next page)

(continued from previous page)

```
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 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    7.522880E+01         0
          P 2          9    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    8.552410E+01         0
          P 2         14    1.160239E+00         0
NOTE: Optimal.
NOTE: Objective = 1.1602389558.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    6.743350E+01         0
          P 2         20    1.029858E+00         0
NOTE: Optimal.
NOTE: Objective = 1.0298577511.
NOTE: The Dual Simplex solve time is 0.01 seconds.
```

(continues on next page)

(continued from previous page)

```
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.119904E+02         0
          P 2          6    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    3.162476E+01         0
          P 2          5    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
```

(continues on next page)

(continued from previous page)

```
         Phase Iteration        Value         Time
          D 2          1    1.462372E+02         0
          P 2          9    1.205868E+00         0
NOTE: Optimal.
NOTE: Objective = 1.2058683067.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.913941E+01         0
          P 2         16    1.228251E+00         0
NOTE: Optimal.
NOTE: Objective = 1.2282514234.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    1.428026E+02         0
          P 2          5    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
```

(continues on next page)

(continued from previous page)

```
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    9.575369E+01         0
          P 2         16    1.213087E+00         0
NOTE: Optimal.
NOTE: Objective = 1.2130872456.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    3.928295E+01         0
          P 2          7    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: Problem generation will use 7 threads.
NOTE: The problem has 29 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (0 LE, 0 EQ, 9 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 0 variables and 0 constraints.
NOTE: The LP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 29 variables, 9 constraints, and 255 constraint␣
↪coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.
                              Objective
         Phase Iteration        Value         Time
          D 2          1    4.906079E+01         0
          P 2          5    1.000000E+00         0
NOTE: Optimal.
NOTE: Objective = 1.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The output table 'EFFICIENCY_DATA' in caslib 'CASUSER(casuser)' has 28 rows and␣
↪3 columns.
NOTE: The output table 'WEIGHT_DATA_DENSE' in caslib 'CASUSER(casuser)' has 17 rows␣
↪and 14 columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The output table 'WEIGHT_DATA_SPARSE' in caslib 'CASUSER(casuser)' has 43 rows␣
↪and 3 columns.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 29 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 9 rows and 4 columns.
Print Table:
    COL1  garage_name  efficiency_number
0    1.0  Bournemouth           1.000000
1    2.0       Henley           1.000000
2    3.0       Woking           0.867320
3    4.0        Alton           1.000000
4    5.0   Dorchester           0.839204
5    6.0    Alresford           1.000000
6    7.0     Ringwood           0.875869
7    8.0   Winchester           0.840174
8    9.0     Weymouth           1.000000
9   10.0     Petworth           0.988237
10  11.0      Reading           0.982052
11  12.0    Weybridge           0.854345
12  13.0   Portsmouth           1.000000
13  14.0      Andover           0.917379
14  15.0      Newbury           1.000000
15  16.0   Maidenhead           1.000000
16  17.0  Basingstoke           1.000000
17  18.0    Salisbury           1.000000
18  19.0        Poole           0.861891
19  20.0     Bridport           0.971008
20  21.0     Portland           1.000000
21  22.0  Petersfield           1.000000
22  23.0     Midhurst           0.829278
23  24.0    Guildford           0.814166
24  25.0       Romsey           1.000000
25  26.0   Chichester           0.824343
26  27.0  Southampton           1.000000
27  28.0      Fareham           1.000000
Efficiency Data:
Selected Rows from Table EFFICIENCY_DATA

    garage  garage_name  efficiency_number
0      1.0  Bournemouth           1.000000
1      2.0       Henley           1.000000
2      3.0       Woking           0.867320
3      4.0        Alton           1.000000
4      5.0   Dorchester           0.839204
5      6.0    Alresford           1.000000
6      7.0     Ringwood           0.875869
7      8.0   Winchester           0.840174
8      9.0     Weymouth           1.000000
9     10.0     Petworth           0.988237
10    11.0      Reading           0.982052
11    12.0    Weybridge           0.854345
12    13.0   Portsmouth           1.000000
13    14.0      Andover           0.917379
14    15.0      Newbury           1.000000
15    16.0   Maidenhead           1.000000
16    17.0  Basingstoke           1.000000
17    18.0    Salisbury           1.000000
```

(continues on next page)

(continued from previous page)

```
18    19.0        Poole           0.861891
19    20.0     Bridport           0.971008
20    21.0     Portland           1.000000
21    22.0  Petersfield           1.000000
22    23.0     Midhurst           0.829278
23    24.0    Guildford           0.814166
24    25.0       Romsey           1.000000
25    26.0   Chichester           0.824343
26    27.0  Southampton           1.000000
27    28.0      Fareham           1.000000
Weight Data (Dense):
Selected Rows from Table WEIGHT_DATA_DENSE

    inefficient_garage  garage_name  efficiency_number  ...       w22  w25       w27
0                  1.0  Bournemouth           1.000000  ...       NaN  NaN       NaN
1                  3.0        Woking          0.867320  ...       NaN  NaN  0.009093
2                  5.0   Dorchester           0.839204  ...       NaN  NaN       NaN
3                  7.0     Ringwood           0.875869  ...       NaN  NaN       NaN
4                  8.0   Winchester           0.840174  ...       NaN  NaN       NaN
5                 10.0     Petworth           0.988237  ...  0.015212  NaN       NaN
6                 11.0      Reading           0.982052  ...       NaN  NaN       NaN
7                 12.0    Weybridge           0.854345  ...       NaN  NaN       NaN
8                 13.0   Portsmouth           1.000000  ...       NaN  NaN       NaN
9                 14.0      Andover           0.917379  ...       NaN  NaN       NaN
10                17.0  Basingstoke           1.000000  ...       NaN  NaN       NaN
11                19.0        Poole           0.861891  ...       NaN  NaN       NaN
12                20.0     Bridport           0.971008  ...       NaN  NaN       NaN
13                23.0     Midhurst           0.829278  ...  0.043482  NaN       NaN
14                24.0    Guildford           0.814166  ...       NaN  NaN       NaN
15                26.0   Chichester           0.824343  ...       NaN  NaN       NaN
16                28.0      Fareham           1.000000  ...       NaN  NaN       NaN

[17 rows x 14 columns]
Weight Data (Sparse):
Selected Rows from Table WEIGHT_DATA_SPARSE

       i     j  weight_sol
0    5.0   2.0    0.035318
1    7.0   2.0    0.146485
2   11.0   2.0    2.862469
3   19.0   2.0    0.434419
4   20.0   2.0    0.783097
5   26.0   2.0    0.236367
6    3.0   4.0    0.021078
7    3.0   6.0    0.952525
8    5.0   6.0    0.104478
9    8.0   6.0    0.416268
10  24.0   6.0    0.622715
11  26.0   6.0    0.096820
12   5.0   9.0    0.119287
13   8.0   9.0    0.333333
14  11.0   9.0    0.544410
15  12.0   9.0    0.796562
16  14.0   9.0    0.857143
17  23.0   9.0    0.066511
18  24.0   9.0    0.191804
19  26.0   9.0    0.335428
```

(continues on next page)

(continued from previous page)

```
20  10.0  15.0    0.066345
21   3.0  16.0    0.148376
22  10.0  16.0    0.034089
23  11.0  16.0    0.137534
24  12.0  16.0    0.145236
25  14.0  16.0    0.214286
26  19.0  16.0    0.344634
27  20.0  16.0    0.194894
28  23.0  16.0    0.008940
29   8.0  18.0    0.403284
30  23.0  18.0    0.059574
31   5.0  21.0    0.751632
32   7.0  21.0    0.319728
33   8.0  21.0    0.096138
34  11.0  21.0    1.199139
35  19.0  21.0    0.757330
36  20.0  21.0    0.469693
37  23.0  21.0    0.471893
38  24.0  21.0    0.168067
39  26.0  21.0    0.165227
40  10.0  22.0    0.015212
41  23.0  22.0    0.043482
42   3.0  27.0    0.009093
Out[8]: 1.0000000000000002
```

4.2 SAS Viya Examples (Abstract)

4.2.1 Curve Fitting

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex11_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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],
```

(continues on next page)

(continued from previous page)

```
        [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
    from sasoptpy.actions import read_data
    POINTS = so.Set(name='POINTS')
    x = so.ParameterGroup(POINTS, name='x')
    y = so.ParameterGroup(POINTS, name='y')
    read_st = read_data(
        table=xy_data,
        index={'target': POINTS, 'key': so.N},
        columns=[
            {'target': x, 'column': 'x'},
            {'target': y, 'column': '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.expr_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.expr_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')

    order.set_init(1)
    L1 = so.Model(name='L1', session=cas_conn)
    L1.set_objective(objective1, sense=so.MIN, name='L1obj')
    L1.include(POINTS, x, y, read_st)
```

(continues on next page)

(continued from previous page)

```
    L1.include(order, beta, estimate, surplus, slack, abs_dev_con)
    L1.add_postsolve_statement('print x y estimate surplus slack;')

    L1.solve(verbose=True)
    if sols:
        sol_data1 = L1.response['Print1.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, name='Linfobj')

    Linf.solve()
    if sols:
        sol_data2 = Linf.response['Print1.PrintTable'].sort_values('x')
        print(so.get_solution_table(beta))
        print(sol_data2.to_string())

    order.set_init(2)

    L1.solve()
    if sols:
        sol_data3 = L1.response['Print1.PrintTable'].sort_values('x')
        print(so.get_solution_table(beta))
        print(sol_data3.to_string())

    Linf.solve()
    if sols:
        sol_data4 = Linf.response['Print1.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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.server_side.curve_fitting import test
```

(continues on next page)

(continued from previous page)

```
In [8]: (s1, s2, s3, s4) = test(cas_conn, sols=True)
NOTE: Cloud Analytic Services made the uploaded file available as table XY_DATA in␣
↪caslib CASUSER(casuser).
NOTE: The table XY_DATA has been created in caslib CASUSER(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 POINTS;
   num x {POINTS};
   num y {POINTS};
   read data XY_DATA into POINTS=[_N_] x y;
   num order init 1;
   var beta {{0..order}};
   impvar estimate {o8 in POINTS} = beta[0] + sum {k in 1..order} (beta[k] * (x[o8]) ^
↪ (k));
   var surplus {{POINTS}} >= 0;
   var slack {{POINTS}} >= 0;
   con abs_dev_con {o32 in POINTS} : y[o32] - estimate[o32] + surplus[o32] -␣
↪slack[o32] = 0;
   min L1obj = sum {i in POINTS} (surplus[i] + slack[i]);
   solve;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.lb␣
↪ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_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 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 0.01 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 40 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 19 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
      beta
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
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 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 41 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 38 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
    beta
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
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 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 0.00 seconds.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 41 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 19 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
```

(continues on next page)

(continued from previous page)

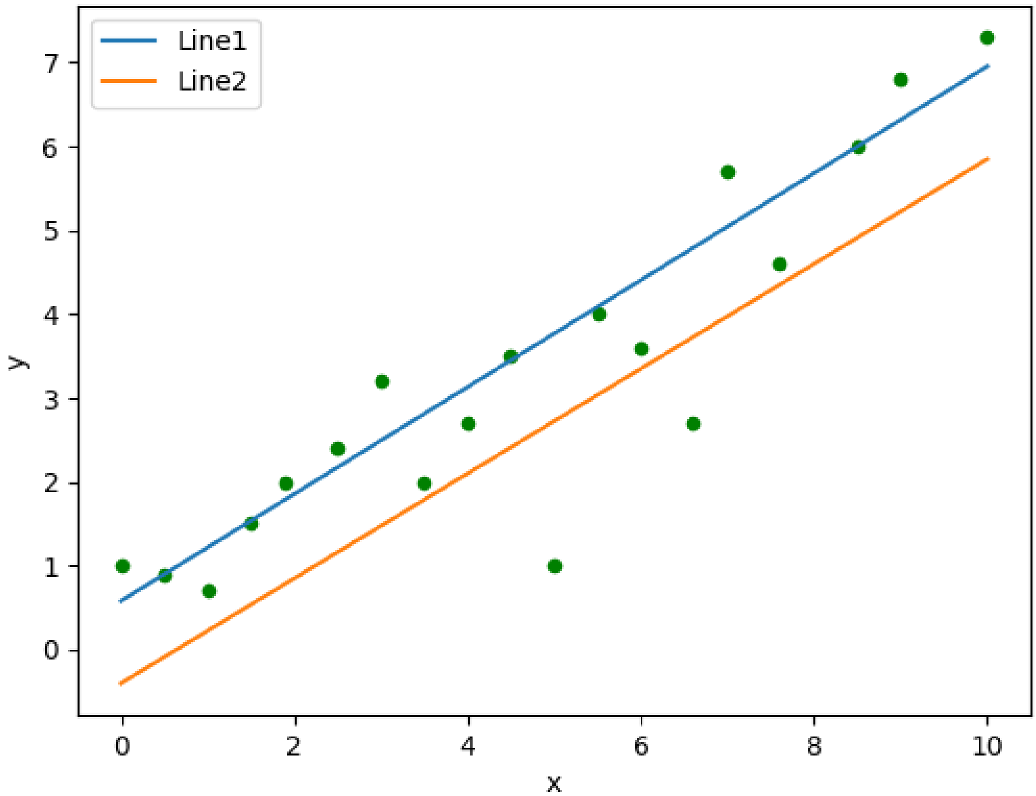
```
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
       beta
0  0.982353
1  0.294510
2  0.033725
    COL1     x    y  estimate       surplus     slack
10  11.0   0.0  1.0  0.982353  0.000000e+00  0.017647
18  19.0   0.5  0.9  1.138039  2.380392e-01  0.000000
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 'CASUSER(casuser)'.
NOTE: Problem generation will use 8 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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 42 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 38 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 13 rows and 4␣
↪columns.
```

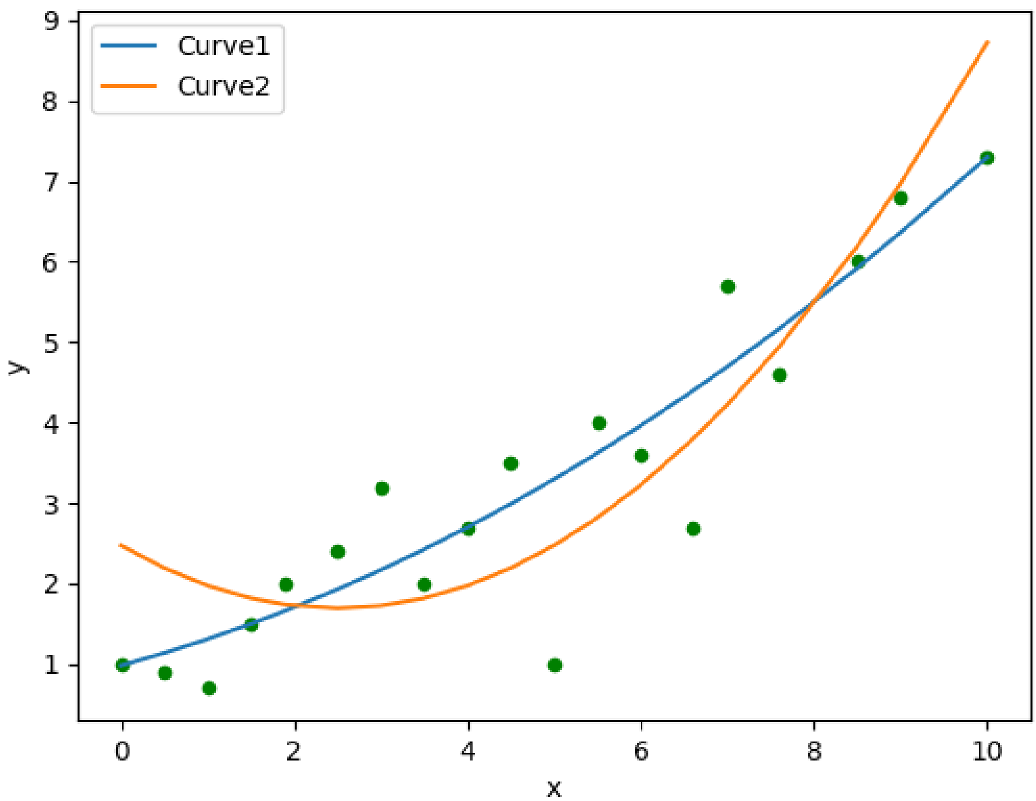
(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 18 rows and 4␣
↪columns.
    beta
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
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 [1]: import matplotlib.pyplot as plt

In [2]: p1 = s1.plot.scatter(x='x', y='y', c='g')

In [3]: s1.plot.line(ax=p1, x='x', y='estimate', label='Line1');

In [4]: s2.plot.line(ax=p1, x='x', y='estimate', label='Line2');

In [5]: p1
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3e1b4e70f0>

In [6]: p2 = s3.plot.scatter(x='x', y='y', c='g')

In [7]: s3.plot.line(ax=p2, x='x', y='estimate', label='Curve1');

In [8]: s4.plot.line(ax=p2, x='x', y='estimate', label='Curve2');

In [9]: p2
Out[9]: <matplotlib.axes._subplots.AxesSubplot at 0x7f3e1b0419b0>
```


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



4.2.2 Nonlinear 1

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_nlpsolver_examples01.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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, name='f1')
```

(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], name='frange')

    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.solve(verbose=True, options={'with': 'nlp', 'algorithm': 'activeset'})

    print(m.get_problem_summary())
    print(m.get_solution_summary())
    if m.get_session_type() == 'CAS':
        print(m.get_solution()[['var', 'value']])

    return m.get_objective_value()
```

Output

```
In [1]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.nonlinear_1 import test

In [8]: 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;
```

(continues on next page)

(continued from previous page)

```
   x[6] = 6;
   x[7] = 1;
   x[8] = 0.5;
   min f1 = 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 frange : -9.9 <= 0.4 * (((x[1]) / (x[7])) ^ (0.67)) + 0.4 * (((x[2]) / (x[8]))␣
↪^ (0.67)) - x[1] - x[2] <= -5.8;
   solve with nlp / algorithm=activeset;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.lb␣
↪ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_CON_.
↪dual;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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        0.18904638
                 3      3.51178229        0.07506389        0.18860455
                 4      4.23595983        0.03595983        0.60088809
                 5      4.16334906                 0        0.47130008
                 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       0.07584723
                18      3.95114031        0.00000941        0.04093117
                19      3.95027690        0.00011307        0.00020755
                20      3.95115797    0.0000007730235       0.00018707
                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
```

(continues on next page)

(continued from previous page)

```
                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 output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 8 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 5 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 20 rows and 4␣
↪columns.
Selected Rows from Table PROBLEMSUMMARY

                                Value
Label
Objective Sense          Minimization
Objective Function                 f1
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
Selected Rows from Table SOLUTIONSUMMARY

                            Value
Label
Solver                        NLP
Algorithm              Active Set
Objective Function             f1
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
Selected Rows from Table SOLUTION

    var     value
0   x[1]  6.463315
```

(continues on next page)

(continued from previous page)

```
1   x[2]   2.234530
2   x[3]   0.667455
3   x[4]   0.595820
4   x[5]   5.932980
5   x[6]   5.527231
6   x[7]   1.013787
7   x[8]   0.400664
Out[8]: 3.951157967716
```

4.2.3 Nonlinear 2

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=ormpug_nlpsolver_examples02.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/nlpse02.html

Model

```python
import sasoptpy as so
import sasoptpy.abstract.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.expr_sum(-4*x[i]+3 for i in so.exp_range(1, N-1)) +
        so.expr_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)
    print(m.get_solution_summary())
    if m.get_session_type() == 'CAS':
        print(m.response['Print1.PrintTable'].head())

    # Model 2
    so.reset()
    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.expr_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]: import os

In [2]: hostname = os.getenv('CASHOST')

In [3]: port = os.getenv('CASPORT')

In [4]: from swat import CAS

In [5]: cas_conn = CAS(hostname, port)

In [6]: import sasoptpy
```

```
In [7]: from examples.client_side.nonlinear_2 import test

In [8]: 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;
   min f = sum {i in 1..N-1} (- 4 * x[i] + 3) + sum {i in 1..N-1} (((x[i]) ^ (2) +␣
↪(x[N]) ^ (2)) ^ (2));
   solve with nlp / ;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.lb␣
↪ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_CON_.
↪dual;
   print x;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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 Direct algorithm is used.
                           Objective                          Optimality
              Iter             Value     Infeasibility             Error
                 0     2997.00000000                 0        2.66666667
                 1      561.93750000                 0        4.44444444
                 2       41.17478400                 0       12.76397516
                 3        0.41230550                 0       43.37933609
                 4        0.00005471                 0        0.47005128
                 5   2.2737367544E-13                0        0.00006316
                 6                 0                 0  1.1527477269E-12
NOTE: Optimal.
NOTE: Objective = 0.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 1000 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 0 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
```

(continues on next page)

(continued from previous page)

```
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
Selected Rows from Table SOLUTIONSUMMARY

                                    Value
Label
Solver                                NLP
Algorithm              Interior Point Direct
Objective Function                      f
Solution Status                   Optimal
Objective Value                         0

Optimality Error             1.152748E-12
Infeasibility                           0

Iterations                              6
Presolve Time                        0.00
Solution Time                        0.02
   COL1    x
0   1.0  1.0
1   2.0  1.0
2   3.0  1.0
3   4.0  1.0
4   5.0  1.0
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 in 1..N-1} (cos(- 0.5 * (x[i + 1]) - ((x[i]) ^ (2))));
   solve with nlp / algorithm=activeset;
   create data solution from [i]= {1.._NVAR_} var=_VAR_.name value=_VAR_ lb=_VAR_.lb␣
↪ub=_VAR_.ub rc=_VAR_.rc;
   create data dual from [j] = {1.._NCON_} con=_CON_.name value=_CON_.body dual=_CON_.
↪dual;
   print x;

NOTE: Submitting OPTMODEL code to CAS server.
NOTE: Problem generation will use 8 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
```

(continues on next page)

(continued from previous page)

```
                  6     -999.00000000                 0  1.7018480129E-12
NOTE: Optimal.
NOTE: Objective = -999.
NOTE: The output table 'SOLUTION' in caslib 'CASUSER(casuser)' has 1000 rows and 6␣
↪columns.
NOTE: The output table 'DUAL' in caslib 'CASUSER(casuser)' has 0 rows and 4 columns.
NOTE: The CAS table 'solutionSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
NOTE: The CAS table 'problemSummary' in caslib 'CASUSER(casuser)' has 12 rows and 4␣
↪columns.
Selected Rows from Table SOLUTIONSUMMARY

                           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               0.05
Out[8]: -999.0
```

4.3 SAS 9.4 Examples

4.3.1 Decentralization (SASPy)

Reference

SAS/OR example: http://go.documentation.sas.com/?docsetId=ormpex&docsetTarget=ormpex_ex10_toc.htm&docsetVersion=15.1&locale=en

SAS/OR code for example: http://support.sas.com/documentation/onlinedoc/or/ex_code/151/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']
```

(continues on next page)

(continued from previous page)

```
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:
            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.expr_sum(benefit[i, j] * assign[i, j]
                           for i in DEPTS for j in CITIES)
```

(continues on next page)

(continued from previous page)

```
    totalCost = so.expr_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.expr_sum(assign[dept, city] for city in CITIES)
                       == 1 for dept in DEPTS), name='assign_dept')

    m.add_constraints((so.expr_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.expr_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.expr_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]: import os

In [2]: import saspy

In [3]: config_file = os.path.abspath('../tests/examples/saspy_config.py')

In [4]: sas_conn = saspy.SASsession(cfgfile=config_file)
Using SAS Config named: sshsas
SAS Connection established. Subprocess id is 158


In [5]: import sasoptpy
```

```
In [6]: from examples.client_side.decentralization import test

In [7]: test(sas_conn)
NOTE: Initialized model decentralization.
NOTE: Converting model decentralization to OPTMODEL.
NOTE: Submitting OPTMODEL code to SAS instance.

NOTE: Writing HTML5(SASPY_INTERNAL) Body file: STDOUT
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      2    -14.9000000    135.0000000  111.04%       0
             0        1      2    -14.9000000     67.5000000  122.07%       0
             0        1      2    -14.9000000     52.0000000  128.65%       0
             0        1      3      8.1000000     52.0000000   84.42%       0
             0        1      3      8.1000000     50.0000000   83.80%       0
             0        1      3      8.1000000     48.2500000   83.21%       0
             0        1      3      8.1000000     40.0000000   79.75%       0
             0        1      3      8.1000000     39.2500000   79.36%       0
             0        1      3      8.1000000     34.2000000   76.32%       0
             0        1      3      8.1000000     33.6187500   75.91%       0
             0        1      3      8.1000000     33.0761905   75.51%       0
             0        1      3      8.1000000     32.6521739   75.19%       0
             0        1      3      8.1000000     32.0142857   74.70%       0
             0        1      3      8.1000000     31.8222222   74.55%       0
             0        1      3      8.1000000     31.3333333   74.15%       0
```

(continues on next page)

(continued from previous page)

```
             0        1      3      8.1000000     30.0000000   73.00%       0
             0        1      3      8.1000000     28.5000000   71.58%       0
             0        1      4     14.9000000     14.9000000    0.00%       0
NOTE: The MILP solver added 28 cuts with 146 cut coefficients at the root.
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.SOLUTION has 105 observations and 6 variables.
NOTE: The data set WORK.DUAL has 278 observations and 4 variables.
NOTE: PROCEDURE OPTMODEL used (Total process time):
      real time           0.18 seconds
      cpu time            0.15 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 code to SAS instance.

NOTE: Writing HTML5(SASPY_INTERNAL) Body file: STDOUT
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.
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.
```

(continues on next page)

(continued from previous page)

```
NOTE: The Branch and Cut algorithm is using up to 4 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      4     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.SOLUTION has 105 observations and 6 variables.
NOTE: The data set WORK.DUAL has 68 observations and 4 variables.
NOTE: PROCEDURE OPTMODEL used (Total process time):
      real time           0.10 seconds
      cpu time            0.08 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
-          80.0       65.1
assign  (A, Bristol)     1.0
        (A, Brighton)    0.0
        (A, London)      0.0
        (B, Bristol)     0.0
        (B, Brighton)    1.0
        (B, London)      0.0
        (C, Bristol)     0.0
        (C, Brighton)    1.0
        (C, London)      0.0
        (D, Bristol)     1.0
        (D, Brighton)    0.0
        (D, London)      0.0
        (E, Bristol)     0.0
        (E, Brighton)    1.0
        (E, London)      0.0
dtype: float64
```

(continues on next page)

(continued from previous page)

```
Out[7]: 14.9
```

CHAPTER

FIVE

API REFERENCE

5.1 Core

5.1.1 Model

Constructor

Model(**kwargs)	Creates an optimization model

sasoptpy.Model

class Model (***kwargs*)

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
```

Components

Model.get_name(self)	Returns model name
Model.set_session(self, session)	Sets the session of model
Model.get_session(self)	Returns the session of the model
Model.get_session_type(self)	Tests whether the model session is defined and still active
Model.set_objective(self, expression, name)	Specifies the objective function for the model
Model.append_objective(self, expression, name)	Appends a new objective to the model
Model.get_objective(self)	Returns the objective function as an Expression object
Model.get_all_objectives(self)	Returns a list of objectives in the model
Model.add_variable(self, name[, vartype, ...])	Adds a new variable to the model
Model.add_variables(self, *argv, name[, ...])	Adds a group of variables to the model
Model.add_implicit_variable(self[, argv, name])	Adds an implicit variable to the model
Model.get_variable(self, name)	Returns the reference to a variable in the model
Model.get_variables(self)	Returns a list of variables
Model.get_grouped_variables(self)	Returns an ordered dictionary of variables
Model.get_implicit_variables(self)	Returns a list of implicit variables
Model.get_variable_coef(self, var)	Returns the objective value coefficient of a variable
Model.drop_variable(self, variable)	Drops a variable from the model
Model.drop_variables(self, *variables)	Drops a variable group from the model
Model.add_constraint(self, c, name)	Adds a single constraint to the model
Model.add_constraints(self, argv, name)	Adds a set of constraints to the model
Model.get_constraint(self, name)	Returns the reference to a constraint in the model
Model.get_constraints(self)	Returns a list of constraints in the model
Model.get_grouped_constraints(self)	Returns an ordered dictionary of constraints
Model.drop_constraint(self, constraint)	Drops a constraint from the model
Model.drop_constraints(self, *constraints)	Drops a constraint group from the model
Model.add_set(self, name[, init, value, settype])	Adds a set to the model
Model.add_parameter(self, *argv, name[, ...])	Adds a abstract.Parameter object to the model
Model.add_statement(self, statement[, ...])	Adds a PROC OPTMODEL statement to the model
Model.get_sets(self)	Returns a list of Set objects in the model
Model.get_parameters(self)	Returns a list of abstract.Parameter and abstract.ParameterGroup objects in the model
Model.get_statements(self)	Returns a list of all statements inside the model
Model.include(self, *argv)	Adds existing variables and constraints to a model

sasoptpy.Model.get_name

`Model.get_name(self)`
Returns model name

sasoptpy.Model.set_session

`Model.set_session(self, session)`
Sets the session of model

Parameters

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

Notes

- You can use CAS sessions (via SWAT package) or SAS sessions (via SASPy package)
- Session of a model can be set at initialization. See *`Model`*.

sasoptpy.Model.get_session

`Model.get_session(self)`
Returns the session of the model

Returns

session [`swat.cas.connection.CAS` or `saspy.SASsession`] Session of the model, or None

sasoptpy.Model.get_session_type

`Model.get_session_type(self)`
Tests whether the model session is defined and still active

Returns

session [string] 'CAS' for CAS sessions, 'SAS' for SAS sessions

sasoptpy.Model.set_objective

`Model.set_objective(self, expression, name, sense=None)`
Specifies the objective function for the model

Parameters

expression [*`Expression`*] The objective function as an Expression

name [string] Name of the objective value

sense [string, optional] Objective value direction, *sasoptpy.MIN* or *sasoptpy.MAX*

Returns

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

See also:

`Model.append_objective()`

Notes

- Default objective sense is minimization *MIN*.
- This method replaces the existing objective of the model. When working with multiple objectives, use the `Model.append_objective()` method.

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')
```

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

sasoptpy.Model.append_objective

`Model.`**`append_objective`**(*self*, *expression*, *name*, *sense=None*)

Appends a new objective to the model

Parameters

expression [`Expression`] The objective function as an Expression

name [string] Name of the objective value

sense [string, optional] Objective value direction, *sasoptpy.MIN* or *sasoptpy.MAX*

Returns

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

See also:

`Model.set_objective()`

Notes

- Default objective sense is minimization *MIN*.

Examples

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

sasoptpy.Model.get_objective

`Model.`**`get_objective`**(*self*)

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_all_objectives

`Model.`**`get_all_objectives`**(*self*)

Returns a list of objectives in the model

Returns

all_objectives [list] A list of *`Objective`* objects

Examples

```
>>> m = so.Model(name='test_set_get_objective')
>>> x = m.add_variable(name='x')
>>> obj1 = m.set_objective(2 * x, sense=so.MIN, name='obj1')
>>> obj2 = m.set_objective(5 * x, sense=so.MIN, name='obj2') # Overrides obj1
>>> obj3 = m.append_objective(10 * x, sense=so.MIN, name='obj3')
>>> assertEqual(m.get_all_objectives(), [obj2, obj3])
True
```

sasoptpy.Model.add_variable

`Model.`**`add_variable`**(*self*, *name*, *vartype=None*, *lb=None*, *ub=None*, *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

name [string] Name of the variable to be created

vartype [string, optional] Type of the variable, either *sasoptpy.BIN*, *sasoptpy.INT* or *sasoptpy.CONT*

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

- *name* is a mandatory field for this method.

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.include(y)
```

sasoptpy.Model.add_variables

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

Adds a group of variables to the model

Parameters

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

name [string] 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()`*

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`**(*self*, *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] Name of the implicit variable

Notes

- Based on whether the implicit variables are generated by a regular or abstract expression, they can 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.get_variable

`Model.`**`get_variable`**(*self*, *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`**(*self*)

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_grouped_variables

`Model.get_grouped_variables(self)`

Returns an ordered dictionary of variables

Returns

grouped_vars [OrderedDict] Dictionary of variables and variable groups in the model

See also:

Model.get_variables(), *Model.get_grouped_constraints()*

Examples

```
>>> m1 = so.Model(name='test_copy_model_1')
>>> x = m1.add_variable(name='x')
>>> y = m1.add_variables(2, name='y')
>>> vars = OrderedDict([('x', x), ('y', y)])
>>> self.assertEqual(m1.get_grouped_variables(), vars)
True
```

sasoptpy.Model.get_implicit_variables

`Model.get_implicit_variables(self)`

Returns a list of implicit variables

Returns

implicit_variables [list] List of implicit variables in the model

Examples

```
>>> m = so.Model(name='test_add_impvar')
>>> x = m.add_variables(5, name='x')
>>> y = m.add_implicit_variable((i * x[i] + x[i] ** 2 for i in range(5)),
                                name='y')
>>> assertEqual([y], m.get_implicit_variables())
True
```

sasoptpy.Model.get_variable_coef

`Model.get_variable_coef(self, var)`

Returns the objective value coefficient of a variable

Parameters

var [`Variable` or string] Variable whose objective value is requested. It can be either the variable object itself, or the name of the variable.

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.drop_variable

`Model.drop_variable(self, 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**(*self*, **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.add_constraint

Model.**add_constraint**(*self*, *c*, *name*)

Adds a single constraint to the model

Parameters

c [*Constraint*] Constraint to be added to the model

name [string] 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**(*self*, *argv*, *name*)
Adds a set of constraints to the model

Parameters

argv [Generator-type object] List of constraints as a generator-type Python object

name [string] 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]
  [(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.get_constraint

`Model.get_constraint(self, name)`

Returns the reference to a constraint in the model

Parameters

name [string] Name of the constraint requested

Returns

constraint [*Constraint*] Requested object

Examples

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

sasoptpy.Model.get_constraints

`Model.get_constraints(self)`

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_grouped_constraints

`Model.get_grouped_constraints(self)`

Returns an ordered dictionary of constraints

Returns

grouped_cons [OrderedDict] Dictionary of constraints and constraint groups in the model

See also:

Model.get_constraints(), *Model.get_grouped_variables()*

Examples

```
>>> m1 = so.Model(name='test_copy_model_1')
>>> x = m1.add_variable(name='x')
>>> y = m1.add_variables(2, name='y')
>>> c1 = m1.add_constraint(x + y[0] >= 2, name='c1')
>>> c2 = m1.add_constraints((x - y[i] <= 10 for i in range(2)), name='c2')
>>> cons = OrderedDict([('c1', c1), ('c2', c2)])
>>> self.assertEqual(m1.get_grouped_constraints(), cons)
True
```

sasoptpy.Model.drop_constraint

Model.**drop_constraint**(*self*, *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**(*self*, **constraints*)

Drops a constraint group from the model

Parameters

constraints [*Constraint* or *ConstraintGroup*] Arbitrary number of constraints to be dropped

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.add_set

`Model.`**`add_set`**(*self*, *name*, *init=None*, *value=None*, *settype=None*)

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 as a list

The list can have one more *num* (for float) and *str* (for string) values. You can use *sasoptpy.NUM* and *sasoptpy.STR* for floats and strings, respectively.

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;
```

```
>>> m.add_set(name='W', settype=[so.STR, so.NUM])
>>> print(W._defn())
set <str, num> W;
```

sasoptpy.Model.add_parameter

`Model.add_parameter(self, *argv, name, init=None, value=None, p_type=None)`

Adds a `abstract.Parameter` object to the model

Parameters

argv [`Set`, optional] Index or indices of the parameter

name [string] 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

Returns

p [`abstract.Parameter` or `abstract.ParameterGroup`] A single parameter or a parameter group

Examples

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

```
>>> I = m.add_set(name='I')
>>> J = m.add_set(name='J')
>>> p = m.add_parameter(I, J, name='p')
>>> print(p._defn())
num p {{I,J}};
```

sasoptpy.Model.add_statement

`Model.add_statement(self, statement, after_solve=None)`

Adds a PROC OPTMODEL statement to the model

Parameters

statement [`Expression` or string] Statement object

after_solve [boolean] Switch for appending the statement after the problem solution

Notes

- If the statement string includes 'print', then the statement is automatically placed after the solve even if *after_solve* is *False*.

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

sasoptpy.Model.get_sets

`Model.get_sets(self)`

Returns a list of `Set` objects in the model

Returns

set_list [list] List of sets in the model

Examples

```
>>> m.get_sets()
[sasoptpy.abstract.Set(name=W, settype=['str', 'num']), sasoptpy.abstract.
↪Set(name=I, settype=['num']), sasoptpy.abstract.Set(name=J, settype=['num'])]
```

sasoptpy.Model.get_parameters

`Model.get_parameters(self)`

Returns a list of *abstract.Parameter* and *abstract.ParameterGroup* objects in the model

Returns

param_list [list] List of parameters in the model

Examples

```
>>> for i in m.get_parameters():
...     print(i.get_name(), type(i))
p <class 'sasoptpy.abstract.parameter_group.ParameterGroup'>
r <class 'sasoptpy.abstract.parameter.Parameter'>
```

sasoptpy.Model.get_statements

`Model.get_statements(self)`

Returns a list of all statements inside the model

Returns

st_list [list] List of all statement objects

Examples

```
>>> m.add_statement(so.abstract.LiteralStatement("expand;"))
>>> m.get_statements()
[<sasoptpy.abstract.statement.literal.LiteralStatement object at 0x7fe0202fc358>]
>>> print(m.to_optmodel())
proc optmodel;
var x;
min obj1 = x * x;
expand;
solve;
quit;
```

sasoptpy.Model.include

`Model.include(self, *argv)`

Adds existing variables and constraints to a model

Parameters

argv : Objects to be included in the model

Notes

- Valid object types for *argv* parameter:
  - `Model`

    Including a model causes all variables and constraints inside the original model to be included.
  - `Variable`
  - `Constraint`
  - `VariableGroup`
  - `ConstraintGroup`
  - `Objective`

- Set
- Parameter
- ParameterGroup
- Statement and all subclasses
- ImplicitVar

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)
```

Solver calls

Model.solve(self, **kwargs)	Solves the model by calling CAS or SAS optimization solvers
Model.tune_parameters(self, **kwargs)	Tunes the model to find ideal solver parameters
Model.get_solution(self[, vtype, solution, …])	Returns the primal and dual problem solutions
Model.get_variable_value(self, var)	Returns the value of a variable
Model.get_objective_value(self)	Returns the optimal objective value
Model.get_solution_summary(self)	Returns the solution summary table to the user
Model.get_problem_summary(self)	Returns the problem summary table to the user
Model.get_tuner_results(self)	Returns the tuning results
Model.print_solution(self)	Prints the current values of the variables
Model.clear_solution(self)	Clears the cached solution of the model

sasoptpy.Model.solve

`Model.solve(self, **kwargs)`

Solves the model by calling CAS or SAS optimization solvers

Parameters

options [dict, optional] Solver options as a dictionary object

submit [boolean, optional] When set to *True*, calls the solver

name [string, optional] Name of the table

frame [boolean, optional] When set to *True*, uploads the problem as a DataFrame in MPS format

drop [boolean, optional] When set to *True*, drops the MPS table after solve (only CAS)

replace [boolean, optional] When set to *True*, replaces an existing MPS table (only CAS and MPS)

primalin [boolean, optional] When set to *True*, uses initial values (only MILP)

verbose [boolean, optional (experimental)] When set to *True*, prints the generated OPTMODEL code

Returns

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

Notes

- Some of the options listed under the `options` argument might not be passed, depending 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.tune_parameters

`Model.tune_parameters(self, **kwargs)`

Tunes the model to find ideal solver parameters

Parameters

kwargs : Keyword arguments as defined in the optimization.tuner action.

Acceptable values are:

- milpParameters: Parameters for the solveMilp action, such as *maxTime*, *heuristics*, *feasTol*
- tunerParameters: Parameters for the tuner itself, such as *maxConfigs*, *printLevel*, *logFreq*
- tuningParameters: List of parameters to be tuned, such as *cutStrategy*, *presolver*, *restarts*

Returns

tunerResults [`swat.dataframe.SASDataFrame`] Tuning results as a table

See also:

`Model.get_tuner_results()`

Notes

- See SAS Optimization documentation for a full list of tunable parameters.
- See Optimization Action Set documentation.

Examples

```
>>> m = so.Model(name='model1')
>>> ...
>>> results = m.tune_parameters(tunerParameters={'maxConfigs': 10})
NOTE: Initialized model knapsack_with_tuner.
NOTE: Added action set 'optimization'.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table KNAPSACK_
↪WITH_TUNER in caslib CASUSER(casuser).
NOTE: The table KNAPSACK_WITH_TUNER has been created in caslib CASUSER(casuser)␣
↪from binary data uploaded to Cloud Analytic Services.
NOTE: Start to tune the MILP
         SolveCalls  Configurations    BestTime        Time
                  1               1        0.21        0.26
                  2               2        0.19        0.50
                  3               3        0.19        0.72
                  4               4        0.19        0.95
                  5               5        0.19        1.17
                  6               6        0.19        1.56
                  7               7        0.18        1.76
                  8               8        0.17        1.96
                  9               9        0.17        2.16
                 10              10        0.17        2.35
NOTE: Configuration limit reached.
NOTE: The tuning time is 2.35 seconds.
```

(continues on next page)

(continued from previous page)

```
>>> print(results)
   Configuration conflictSearch  ... Sum of Run Times Percentage Successful
0            0.0      automatic  ...             0.20                 100.0
1            1.0           none  ...             0.17                 100.0
2            2.0           none  ...             0.17                 100.0
3            3.0       moderate  ...             0.17                 100.0
4            4.0           none  ...             0.18                 100.0
5            5.0           none  ...             0.18                 100.0
6            6.0     aggressive  ...             0.18                 100.0
7            7.0       moderate  ...             0.18                 100.0
8            8.0     aggressive  ...             0.19                 100.0
9            9.0      automatic  ...             0.36                 100.0
```

```
>>> results = m.tune_parameters(
       milpParameters={'maxtime': 10},
       tunerParameters={'maxConfigs': 20, 'logfreq': 5},
       tuningParameters=[
          {'option': 'presolver', 'initial': 'none', 'values': ['basic',
↪'aggressive', 'none']},
          {'option': 'cutStrategy'},
          {'option': 'strongIter', 'initial': -1, 'values': [-1, 100, 1000]}
       ])
NOTE: Added action set 'optimization'.
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table KNAPSACK_
↪WITH_TUNER in caslib CASUSER(casuser).
NOTE: The table KNAPSACK_WITH_TUNER has been created in caslib CASUSER(casuser)␣
↪from binary data uploaded to Cloud Analytic Services.
NOTE: Start to tune the MILP
         SolveCalls  Configurations    BestTime        Time
                  5               5        0.17        1.01
                 10              10        0.17        2.00
                 15              15        0.17        2.98
                 20              20        0.17        3.95
NOTE: Configuration limit reached.
NOTE: The tuning time is 3.95 seconds.
>>> print(results)
    Configuration conflictSearch  ... Sum of Run Times Percentage Successful
0             0.0      automatic  ...             0.17                 100.0
1             1.0           none  ...             0.16                 100.0
2             2.0           none  ...             0.16                 100.0
3             3.0           none  ...             0.16                 100.0
4             4.0           none  ...             0.16                 100.0
5             5.0           none  ...             0.17                 100.0
6             6.0           none  ...             0.17                 100.0
7             7.0           none  ...             0.17                 100.0
8             8.0           none  ...             0.17                 100.0
9             9.0           none  ...             0.17                 100.0
10           10.0           none  ...             0.17                 100.0
11           11.0     aggressive  ...             0.17                 100.0
12           12.0           none  ...             0.17                 100.0
13           13.0     aggressive  ...             0.17                 100.0
14           14.0      automatic  ...             0.17                 100.0
15           15.0           none  ...             0.17                 100.0
16           16.0           none  ...             0.17                 100.0
17           17.0       moderate  ...             0.17                 100.0
```

(continues on next page)

(continued from previous page)

```
18              18.0        moderate  ...          0.17              100.0
19              19.0            none  ...          0.17              100.0
```

sasoptpy.Model.get_solution

Model.**get_solution**(*self*, *vtype='Primal'*, *solution=None*, *pivot=False*)
Returns the primal and dual problem solutions

Parameters

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

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

pivot [boolean, optional] When set to *True*, returns multiple solutions in columns as a pivot table

Returns

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

Notes

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

(continues on next page)

(continued from previous page)

```
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`**(*self*, *var*)

Returns the value of a variable

Parameters

var [`Variable` or string] Variable reference

Notes

- It is possible to get a variable's value by using the `Variable.get_value()` method, as long as 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(self)`

Returns the optimal objective value

Returns

objective_value [float] Optimal objective value at current solution

Notes

- This method should be used for getting the objective value after solve.
- In order to get the current value of the objective after changing variable values, you can use `m.get_objective().get_value()`.

Examples

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

sasoptpy.Model.get_solution_summary

`Model.get_solution_summary(self)`

Returns the solution summary table to the user

Returns

ss [`swat.dataframe.SASDataFrame`] Solution summary table, that is obtained after `Model.solve()`

Examples

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

```
>>> print(soln)
Solution Summary
                               Value
Label
Solver                            LP
```

(continues on next page)

(continued from previous page)

```
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**(*self*)
Returns the problem summary table to the user

Returns

ps [swat.dataframe.SASDataFrame] Problem summary table, that is obtained after *Model.solve()*

Examples

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

```
>>> print(ps)
Problem Summary
                                Value
Label
Problem Name                   model1
Objective Sense          Maximization
Objective Function                obj
RHS                               RHS
Number of Variables                 2
Bounded Above                       0
Bounded Below                       2
Bounded Above and Below             0
Free                                0
Fixed                               0
Number of Constraints               2
```

(continues on next page)

(continued from previous page)

```
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.get_tuner_results

Model.**get_tuner_results**(*self*)

Returns the tuning results

Returns

tunerResults [dict] Returns tuner results as a dictionary.

Its members are

- Performance Information
- Tuner Information
- Tuner Summary
- Tuner Results

See also:

Model.tune_parameters()

Examples

```
>>> m.tune_parameters(tunerParameters={'maxConfigs': 10})
>>> results = m.get_tuner_reults()
```

sasoptpy.Model.print_solution

`Model.print_solution(self)`

Prints the current values of the variables

See also:

`Model.get_solution()`

Notes

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

Examples

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

sasoptpy.Model.clear_solution

`Model.clear_solution(self)`

Clears the cached solution of the model

Notes

- This method cleans the optimal objective value and solution time parameters of the model.

Export

`Model.to_mps`(self, **kwargs)	Returns the problem in MPS format
`Model.to_optmodel`(self, **kwargs)	Returns the model in OPTMODEL format

sasoptpy.Model.to_mps

`Model.to_mps(self, **kwargs)`

Returns the problem in MPS format

Examples

```
>>> print(n.to_mps())
    Field1 Field2 Field3  Field4 Field5  Field6  _id_
0     NAME              n     0.0           0.0     1
1     ROWS                    NaN           NaN     2
2      MIN  myobj             NaN           NaN     3
3  COLUMNS                    NaN           NaN     4
4               y   myobj     2.0           NaN     5
5      RHS                    NaN           NaN     6
6   RANGES                    NaN           NaN     7
7   BOUNDS                    NaN           NaN     8
8       FR    BND       y     NaN           NaN     9
9   ENDATA                    0.0           0.0    10
```

sasoptpy.Model.to_optmodel

`Model.to_optmodel`(*self*, ***kwargs*)

Returns the model in OPTMODEL format

Examples

```
>>> print(n.to_optmodel())
proc optmodel;
var y init 2;
min myobj = 2 * y;
solve;
quit;
```

Internal functions

Model._is_linear(self)	Checks whether the model can be written as a linear model (in MPS format)

sasoptpy.Model._is_linear

`Model._is_linear`(*self*)

Checks whether 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

Deprecated

Deprecated since version 1.0.0.

The following method(s) are deprecated and will be removed in future minor updates.

`Model.to_frame`(self, **kwargs)	

sasoptpy.Model.to_frame

`Model.to_frame`(*self*, ***kwargs*)

5.1.2 Expression

Constructor

`Expression`([exp, name])	Creates a mathematical expression to represent model components
`Auxiliary`(base[, prefix, suffix, operator, ...])	Represents an auxiliary expression, often as a symbolic attribute
`Symbol`(name)	Represents a symbolic string, to be evaluated on server-side

sasoptpy.Expression

class Expression(*exp=None*, *name=None*)

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

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

sasoptpy.Auxiliary

class Auxiliary(*base*, *prefix=None*, *suffix=None*, *operator=None*, *value=None*)

Bases: `sasoptpy.core.expression.Expression`

Represents an auxiliary expression, often as a symbolic attribute

Parameters

base [*Expression*] Original owner of the auxiliary value

prefix [string, optional] Prefix of the expression

suffix [string, optional] Suffix of the expression

operator [string, optional] Wrapping operator

value [float, optional] Initial value of the symbolic object

Notes

- Auxiliary objects are for internal use

sasoptpy.Symbol

class Symbol(*name*)

Bases: `sasoptpy.core.expression.Expression`

Represents a symbolic string, to be evaluated on server-side

Parameters

name [string] String to be symbolized

Notes

- A Symbol object can be used for any values that does not translate to a value on client-side, but has meaning on execution. For example, *_N_* is a SAS symbol, which can be used in PROC OPTMODEL strings.

General methods

Expression.set_name(self[, name])	Specifies the name of the expression
Expression.set_permanent(self)	Converts a temporary expression into a permanent one
Expression.set_temporary(self)	Converts expression into a temporary expression to enable in-place operations
Expression.get_name(self)	Returns the name of the object
Expression.get_value(self)	Calculates and returns the value of the linear expression
Expression.get_dual(self)	Returns the dual value

sasoptpy.Expression.set_name

Expression.**set_name**(*self*, *name=None*)

Specifies 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(self)`

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

sasoptpy.Expression.set_temporary

`Expression.set_temporary(self)`

Converts expression into a temporary expression to enable in-place operations

sasoptpy.Expression.get_name

`Expression.get_name(self)`

Returns the name of the object

Returns

name [string] Name of the object

Examples

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

sasoptpy.Expression.get_value

`Expression.get_value(self)`

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.get_dual

Expression.**get_dual**(*self*)

Returns the dual value

Returns

dual [float] Dual value of the object

Operations

Expression.add(self, other[, sign])	Combines two expressions and produces a new one
Expression.copy(self[, name])	Returns a copy of the *Expression* object
Expression.mult(self, other)	Multiplies the *Expression* by a scalar value
Expression.get_member(self, key)	Returns the requested member of the expression
Expression.get_member_dict(self)	Returns an ordered dictionary of elements
Expression.get_member_value(self, key)	Returns coefficient of requested member
Expression.get_constant(self)	Returns the constant term in the expression
Expression.set_member(self, key, ref, val[, op])	Adds a new member or changes an existing member
Expression.set_member_value(self, key, value)	Changes the coefficient of the requested member
Expression.add_to_member_value(self, key, value)	Adds *value* to the coefficient of the requested member
Expression.mult_member_value(self, key, value)	Multiplies the coefficient of the requested member by the specified *value*
Expression.copy_member(self, key, exp)	Copies the member of another expression
Expression.delete_member(self, key)	Deletes the requested member from the core dictionary

sasoptpy.Expression.add

Expression.**add**(*self*, *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

- It is preferable to use regular Python operation, instead of calling this method:

```
>>> e = x - y
>>> f = 3 * x + 2 * y
>>> g = e + f
>>> print(g)
4 * x + y
```

sasoptpy.Expression.copy

`Expression.copy(self, 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.mult

`Expression.mult(self, other)`

Multiplies the `Expression` by 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.
- It is preferable to use regular Python operation, instead of calling this method:

```
>>> e = 3 * (x-y)
>>> f = 3
>>> g = e*f
>>> print(g)
9 * x - 9 * y
```

sasoptpy.Expression.get_member

Expression.**get_member**(*self*, *key*)
Returns the requested member of the expression

Parameters

key [string] Identifier of the member, name for single objects

Returns

member [dict] A dictionary of coefficient, operator, and reference of member

sasoptpy.Expression.get_member_dict

Expression.**get_member_dict**(*self*)
Returns an ordered dictionary of elements

sasoptpy.Expression.get_member_value

Expression.**get_member_value**(*self*, *key*)
Returns coefficient of requested member

Parameters

key [string] Identifier of the member

Returns

value [float] Coefficient value of the requested member

sasoptpy.Expression.get_constant

Expression.**get_constant**(*self*)
Returns the constant term in the expression

Examples

```
>>> x = so.Variable(name='x')
>>> e = 2 * x + 5
>>> print(e.get_constant())
5
```

sasoptpy.Expression.set_member

Expression.**set_member**(*self*, *key*, *ref*, *val*, *op=None*)
Adds a new member or changes an existing member

Parameters

key [string] Identifier of the new or existing member

ref [Object] A reference to the new or existing member

val [float] Initial coefficient of the new or existing member

op [string, optional] Operator, if member has multiple children

sasoptpy.Expression.set_member_value

`Expression.`**`set_member_value`**(*self*, *key*, *value*)

Changes the coefficient of the requested member

Parameters

key [string] Identifier of the member

value [float] New coefficient value of the member

sasoptpy.Expression.add_to_member_value

`Expression.`**`add_to_member_value`**(*self*, *key*, *value*)

Adds *value* to the coefficient of the requested member

Parameters

key [string] Identifier of the member

value [float] Value to be added

sasoptpy.Expression.mult_member_value

`Expression.`**`mult_member_value`**(*self*, *key*, *value*)

Multiplies the coefficient of the requested member by the specified *value*

Parameters

key [string] Identifier of the member

value [float] Value to be multiplied with

sasoptpy.Expression.copy_member

`Expression.`**`copy_member`**(*self*, *key*, *exp*)

Copies the member of another expression

Parameters

key [string] Identifier of the member

exp [`Expression`] Other expression to be copied from

sasoptpy.Expression.delete_member

`Expression.delete_member(self, key)`

Deletes the requested member from the core dictionary

Class methods

`Expression.to_expression(obj)`	

sasoptpy.Expression.to_expression

`classmethod Expression.to_expression(obj)`

Private Methods

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

sasoptpy.Expression._expr

`Expression._expr(self)`

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(self)`

Checks whether 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(self, other, direction_)`

Creates a logical relation between `Expression` objects

Parameters

other [`Expression`] Expression on the other side of the relation with respect to 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__(self)`

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__**(*self*)

Generates a representation string that is Python-compatible

Examples

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

5.1.3 Objective

Constructor

Objective(**kwargs)	Objective represents expressions with sense and used as target value in optimization

sasoptpy.Objective

class Objective(***kwargs*)

Bases: sasoptpy.core.expression.Expression

Objective represents expressions with sense and used as target value in optimization

Parameters

exp [*Expression*] Objective as an expression

name [string] Unique name of the expression

sense [string, optional] Direction of the objective, sasoptpy.MIN (default) or sasoptpy.MAX

Examples

```
>>> m = so.Model(name='test_objective')
>>> x = m.add_variable(name='x')
>>> obj = m.set_objective(2 * x - x ** 3, sense=so.MIN, name='new_obj')
>>> str(m.get_objective())
2 * x - (x) ** (3)
>>> type(obj)
sasoptpy.Objective
```

Methods

Objective.set_sense(self, sense)	Specifies the objective sense (direction)
Objective.get_sense(self)	Returns the objective sense (direction)

sasoptpy.Objective.set_sense

`Objective.set_sense(self, sense)`
Specifies the objective sense (direction)

Parameters

sense [string] sasoptpy.MIN or sasoptpy.MAX

sasoptpy.Objective.get_sense

`Objective.get_sense(self)`
Returns the objective sense (direction)

5.1.4 Variable

Constructor

Variable(**kwargs)	Creates an optimization variable to be used inside models

sasoptpy.Variable

class Variable(***kwargs*)
Bases: `sasoptpy.core.expression.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] When set to *True*, indicates that the variable is abstract

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')
```

Attributes

lb Lower bound of the variable

ub Upper bound of the variable

Methods

Variable.set_bounds(self, *[, lb, ub])	Changes bounds on a variable
Variable.set_init(self[, init])	Changes initial value of a variable
Variable.get_type(self)	Returns the type of variable
Variable.get_attributes(self)	Returns an ordered dictionary of main attributes

sasoptpy.Variable.set_bounds

Variable.**set_bounds**(*self*, *, *lb=None*, *ub=None*)

Changes bounds on a variable

Parameters

lb [float or *Expression*] Lower bound of the variable

ub [float or *Expression*] 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(self, init=None)`

Changes initial value of a variable

Parameters

init [float or None] Initial value of the variable

Examples

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

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

sasoptpy.Variable.get_type

`Variable.get_type(self)`

Returns the type of variable

Valid values are:

- *sasoptpy.CONT*
- *sasoptpy.INT*
- *sasoptpy.BIN*

sasoptpy.Variable.get_attributes

`Variable.get_attributes(self)`

Returns an ordered dictionary of main attributes

Returns

attributes [OrderedDict] Dictionary consists of *init*, *lb*, and *ub* attributes

Inherited Methods

`Variable.copy`(self[, name])	Returns a copy of the `Expression` object
`Variable.get_dual`(self)	Returns the dual value
`Variable.get_name`(self)	Returns the name of the object
`Variable.get_value`(self)	Returns the value of the variable

sasoptpy.Variable.copy

`Variable.copy(self, 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(self)`

Returns the dual value

Returns

dual [float] Dual value of the object

sasoptpy.Variable.get_name

`Variable.get_name(self)`

Returns the name of the object

Returns

name [string] Name of the object

Examples

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

sasoptpy.Variable.get_value

`Variable.get_value(self)`

Returns the value of the variable

Returns

value [float] Value of the variable

Examples

```
>>> x.set_value(20)
>>> x.get_value()
20
```

5.1.5 Variable Group

Constructor

`VariableGroup(**kwargs)`	Creates a group of `Variable` objects

sasoptpy.VariableGroup

class VariableGroup(***kwargs*)

Bases: `sasoptpy.core.group.Group`

Creates a group of `Variable` objects

Parameters

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

name [string, optional] Name (prefix) of the variables

vartype [string, optional] Type of variables, *BIN*, *INT*, or *CONT*

lb [list, dict, `pandas.Series`, optional] Lower bounds of variables

ub [list, dict, `pandas.Series`, optional] Upper bounds of variables

init [float, optional] Initial values of variables

See also:

`sasoptpy.Model.add_variables()`

`sasoptpy.Model.include()`

Notes

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

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

Examples

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

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

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

Methods

VariableGroup.get_name(self)	Returns the name of the variable group
VariableGroup.get_attributes(self)	Returns an ordered dictionary of main attributes
VariableGroup.get_type(self)	Returns the type of variable
VariableGroup.get_members(self)	Returns a dictionary of members
VariableGroup.set_bounds(self[, lb, ub, members])	Specifies or updates bounds for the variable group
VariableGroup.set_init(self, init)	Specifies or updates the initial values
VariableGroup.mult(self, vector)	Quick multiplication method for the variable groups
VariableGroup.sum(self, *argv)	Quick sum method for the variable groups

sasoptpy.VariableGroup.get_name

VariableGroup.**get_name**(*self*)

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.get_attributes

VariableGroup.**get_attributes**(*self*)

Returns an ordered dictionary of main attributes

Returns

attributes [OrderedDict] The dictionary consists of *init*, *lb*, and *ub* attributes

sasoptpy.VariableGroup.get_type

VariableGroup.**get_type**(*self*)

Returns the type of variable

Possible values are:

- sasoptpy.CONT
- sasoptpy.INT
- sasoptpy.BIN

Examples

```
>>> z = so.VariableGroup(3, name='z', vartype=so.INT)
>>> z.get_type()
'INT'
```

sasoptpy.VariableGroup.get_members

`VariableGroup.`**`get_members`**`(`*self*`)`

Returns a dictionary of members

sasoptpy.VariableGroup.set_bounds

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

Specifies or updates bounds for the variable group

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`**`(`*self*, *init*`)`

Specifies or updates the initial values

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` (*self*, *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 specified vector

Examples

Multiplying with a list

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

Multiplying with a dictionary

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

Multiplying with a pandas.Series object

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

Multiplying with a pandas.DataFrame object

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

(continues on next page)

(continued from previous page)

```
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`**(*self*, **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']
```

5.1.6 Constraint

Constructor

Constraint(**kwargs)	Creates a linear or quadratic constraint for optimization models

sasoptpy.Constraint

class Constraint(***kwargs*)

Bases: `sasoptpy.core.expression.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, optional] Direction of the logical expression

Possible values are

- *E* for equality (=) constraints
- *L* for less than or euqal to (<=) constraints
- *G* for greater than or equal to (>=) constraints

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 two different ways:
  - 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')
```

  - 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')
```

Methods

`Constraint.get_value`(self[, rhs])	Returns the current value of the constraint
`Constraint.get_dual`(self)	Returns the dual value if exists
`Constraint.set_block`(self, block_number)	Sets the decomposition block number for a constraint
`Constraint.set_direction`(self, direction)	Changes the direction of a constraint
`Constraint.set_rhs`(self, value)	Changes the constant value (right-hand side) of a constraint
`Constraint.update_var_coef`(self, var, value)	Updates the coefficient of a variable inside the constraint

sasoptpy.Constraint.get_value

`Constraint.`**`get_value`**`(`*self*, *rhs=False*`)`

Returns the current value of the constraint

Parameters

rhs [boolean, optional] When set to *True*, includes constant values to the value of the constraint. 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.get_dual

`Constraint.`**`get_dual`**(*self*)

Returns the dual value if exists

Returns

dual [float] Dual value of the constraint

sasoptpy.Constraint.set_block

`Constraint.`**`set_block`**(*self*, *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`**(*self*, *direction*)

Changes the direction of a constraint

Parameters

direction [string] Direction of the constraint

Possible values are

- *E* for equality (=) constraints
- *L* for less than or euqal to (<=) constraints
- *G* for greater than or equal to (>=) constraints

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(self, value)`

Changes the constant value (right-hand side) of a constraint

Parameters

value [float] New right-hand side value for the constraint

Examples

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

sasoptpy.Constraint.update_var_coef

`Constraint.update_var_coef(self, var, value)`

Updates the coefficient of a variable inside the constraint

Parameters

var [`Variable`] Variable to be updated

value [float] Coefficient of the variable in the constraint

See also:

sasoptpy.Model.set_coef()

Examples

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

5.1.7 Constraint Group

Constructor

`ConstraintGroup`(**kwargs)	Creates a group of `Constraint` objects

sasoptpy.ConstraintGroup

class ConstraintGroup(***kwargs*)

Bases: `sasoptpy.core.group.Group`

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]
]
```

Methods

ConstraintGroup.get_name(self)	Returns the name of the constraint group
ConstraintGroup.get_all_keys(self)	Returns a list of all keys (indices) in the group
ConstraintGroup.get_expressions(self[, rhs])	Returns constraints as a list of expressions
ConstraintGroup.get_members(self)	Returns a dictionary of members

sasoptpy.ConstraintGroup.get_name

ConstraintGroup.**get_name**(*self*)

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']
>>> 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_all_keys

ConstraintGroup.**get_all_keys**(*self*)

Returns a list of all keys (indices) in the group

sasoptpy.ConstraintGroup.get_expressions

ConstraintGroup.**get_expressions**(*self*, *rhs=False*)

Returns constraints as a list of expressions

Parameters

rhs [boolean, optional] When set to *True*, passes the the constant part (rhs) of the constraint

Returns

df [pandas.DataFrame] Returns a DataFrame that consists 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
```

sasoptpy.ConstraintGroup.get_members

ConstraintGroup.**get_members**(*self*)

Returns a dictionary of members

5.1.8 Workspace

Constructor

Workspace(name[, session])	Workspace represents an OPTMODEL block that allows multiple solves

sasoptpy.Workspace

class Workspace(*name*, *session=None*)

Bases: object

Workspace represents an OPTMODEL block that allows multiple solves

Parameters

name [string] Name of the workspace

session [saspy.SASsession or swat.cas.connection.CAS, optional] Session to be submitted

Methods

`Workspace.get_elements`(self)	Returns a list of elements in the workspace
`Workspace.set_active_model`(self, model)	Marks the specified model as active; to be used in solve statements
`Workspace.append`(self, element)	Appends a new element (operation or statement) to the workspace
`Workspace.submit`(self, **kwargs)	Submits the workspace as an OPTMODEL block and returns solutions
`Workspace.parse_solve_responses`(self)	Retrieves the solutions to all solve statements
`Workspace.parse_print_responses`(self)	Retrieves responses to all print statements
`Workspace.get_variable`(self, name)	Obtains the value of a specified variable name
`Workspace.set_variable_value`(self, name, value)	Specifies the value of a variable
`Workspace.to_optmodel`(self)	Returns equivalent OPTMODEL code of the workspace

sasoptpy.Workspace.get_elements

`Workspace.`**`get_elements`**(*self*)

Returns a list of elements in the workspace

sasoptpy.Workspace.set_active_model

`Workspace.`**`set_active_model`**(*self*, *model*)

Marks the specified model as active; to be used in solve statements

Parameters

model [`Model`] Model to be activated

sasoptpy.Workspace.append

`Workspace.`**`append`**(*self*, *element*)

Appends a new element (operation or statement) to the workspace

Parameters

element [`sasoptpy.abstract.Statement`] Any statement that can be appended

sasoptpy.Workspace.submit

`Workspace.`**`submit`**(*self*, ***kwargs*)

Submits the workspace as an OPTMODEL block and returns solutions

sasoptpy.Workspace.parse_solve_responses

`Workspace.parse_solve_responses(self)`
Retrieves the solutions to all solve statements

sasoptpy.Workspace.parse_print_responses

`Workspace.parse_print_responses(self)`
Retrieves responses to all print statements

sasoptpy.Workspace.get_variable

`Workspace.get_variable(self, name)`
Obtains the value of a specified variable name

Parameters

name [string] Name of the variable

sasoptpy.Workspace.set_variable_value

`Workspace.set_variable_value(self, name, value)`
Specifies the value of a variable

Parameters

name [string] Name of the variable

value [float] New value of the variable

sasoptpy.Workspace.to_optmodel

`Workspace.to_optmodel(self)`
Returns equivalent OPTMODEL code of the workspace

Returns

optmodel [string] Generated OPTMODEL code of the workspace object

5.2 Abstract

5.2.1 Abstract

Main classes

Parameter(**kwargs)	Represents a problem input parameter
ParameterGroup(**kwargs)	Represents a group of input parameters
Set(**kwargs)	Creates an index set to be represented inside PROC OPTMODEL

Continued on next page

Table 25 – continued from previous page

SetIterator(initset[, name, datatype])	Creates an iterator object for a given Set
SetIteratorGroup(initset[, datatype, names])	Creates a group of set iterator objects for multi-dimensional sets
Statement()	Creates a statement to be executed at the server
ImplicitVar(**kwargs)	Creates an implicit variable

sasoptpy.abstract.Parameter

class Parameter(***kwargs*)

Bases: `sasoptpy.core.expression.Expression`

Represents a problem input parameter

Parameters

name [string] Name of the parameter

ptype [string, optional] Type of the parameter. Possible values are *sasoptpy.STR* and *sasoptpy.NUM*

value [float, optional] Value of the parameter

init [float, optional] Initial value of the parameter

Examples

```
>>> with so.Workspace('w') as w:
...     p = so.Parameter(name='p', init=3)
...     p.set_value(5)
...
<sasoptpy.abstract.statement.assignment.Assignment object at 0x7f7952e9bb38>
>>> print(so.to_optmodel(w))
proc optmodel;
   num p init 3;
   p = 5;
quit;
```

sasoptpy.abstract.ParameterGroup

class ParameterGroup(***kwargs*)

Bases: `object`

Represents a group of input parameters

Parameters

index_key [iterable] Index key of the group members

name [string] Name of the parameter group

ptype [string, optional] Type of the parameters. Possible values are *sasoptpy.STR* and *sasoptpy.NUM*

value [float, optional] Value of the parameter

init [float, optional] Initial value of the parameter

Examples

```
>>> from sasoptpy.actions import for_loop
>>> with so.Workspace('w') as w:
...     p = so.ParameterGroup(so.exp_range(1, 6), name='p', init=3)
...     p[0].set_value(3)
...     S = so.Set(name='S', value=so.exp_range(1, 6))
...     for i in for_loop(S):
...         p[i].set_value(1)
...
>>> print(so.to_optmodel(w))
proc optmodel;
   num p {1..5} init 3;
   p[0] = 3;
   set S = 1..5;
   for {o13 in S} do;
      p[o13] = 1;
   end;
quit;
```

sasoptpy.abstract.Set

class Set(***kwargs*)

Bases: `object`

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

sasoptpy.abstract.SetIterator

class SetIterator(*initset*, *name=None*, *datatype=None*)

Bases: `sasoptpy.core.expression.Expression`

Creates an iterator object for a given Set

Parameters

initset [*Set*] Set to be iterated on

name [string, optional] Name of the iterator

datatype [string, optional] Type of the iterator

Notes

- `abstract.SetIterator` objects are created automatically when iterating over a `abstract.Set` object

Examples

```
>>> S = so.Set(name='S')
>>> for i in S:
...     print(i.get_name(), type(i))
o19 <class 'sasoptpy.abstract.set_iterator.SetIterator'>
```

sasoptpy.abstract.SetIteratorGroup

class SetIteratorGroup(*initset*, *datatype=None*, *names=None*)

Bases: `collections.OrderedDict`, `sasoptpy.core.expression.Expression`

Creates a group of set iterator objects for multi-dimensional sets

Parameters

initset [*Set*] Set to be iterated on

names [string, optional] Names of the iterators

datatype [string, optional] Types of the iterators

Examples

```
>>> T = so.Set(name='T', settype=[so.STR, so.NUM])
>>> for j in T:
...     print(j.get_name(), type(j))
...     for k in j:
...         print(k.get_name(), type(k))
o5 <class 'sasoptpy.abstract.set_iterator.SetIteratorGroup'>
o6 <class 'sasoptpy.abstract.set_iterator.SetIterator'>
o8 <class 'sasoptpy.abstract.set_iterator.SetIterator'>
```

sasoptpy.abstract.Statement

class Statement

Bases: `abc.ABC`

Creates a statement to be executed at the server

This class is an abstract base class for all statement types.

sasoptpy.abstract.ImplicitVar

class ImplicitVar(***kwargs*)

Bases: `object`

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.abstract.SetIterator(name=i_1, ...),) x + i_1 * y[i_1]
```

Statements

The following list of classes define the underlying structure for the abstract functions. See *Abstract Actions* to see how you can use abstract functions and statements.

`Assignment`(identifier, expression[, keyword])	
`CoForLoopStatement`(*args)	
`CreateDataStatement`(table, index[, columns])	
`DropStatement`(**kwargs)	
`ForLoopStatement`(*args)	
`IfElseStatement`(logic_expression, if_statement)	
`LiteralStatement`(**kwargs)	
`ObjectiveStatement`(expression, **kwargs)	
`ReadDataStatement`(table, index[, columns])	
`SolveStatement`(*args, **kwargs)	
`FixStatement`(*elements)	
`UnfixStatement`(*elements)	
`PrintStatement`(*args)	

sasoptpy.abstract.statement.Assignment

class Assignment(*identifier*, *expression*, *keyword=None*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.CoForLoopStatement

class CoForLoopStatement(**args*)
Bases: `sasoptpy.abstract.statement.for_loop.ForLoopStatement`

sasoptpy.abstract.statement.CreateDataStatement

class CreateDataStatement(*table*, *index*, *columns=None*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.DropStatement

class DropStatement(***kwargs*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.ForLoopStatement

class ForLoopStatement(**args*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.IfElseStatement

class IfElseStatement(*logic_expression*, *if_statement*, *else_statement=None*)
Bases: `sasoptpy.abstract.statement.if_else.NestedConditions`

sasoptpy.abstract.statement.LiteralStatement

class LiteralStatement(***kwargs*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.ObjectiveStatement

class ObjectiveStatement(*expression*, ***kwargs*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.ReadDataStatement

class ReadDataStatement(*table*, *index*, *columns=None*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.SolveStatement

class SolveStatement(**args*, ***kwargs*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.FixStatement

class FixStatement(**elements*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.UnfixStatement

class UnfixStatement(**elements*)
Bases: `sasoptpy.abstract.statement.statement_base.Statement`

sasoptpy.abstract.statement.PrintStatement

class PrintStatement(**args*)

Bases: `sasoptpy.abstract.statement.statement_base.Statement`

5.3 Interface

5.3.1 Interface

CAS (Viya)

CASMediator(caller, cas_session)	Handles the connection between sasoptpy and the SAS Viya (CAS) server

sasoptpy.interface.CASMediator

class CASMediator(*caller*, *cas_session*)

Bases: `sasoptpy.interface.solver.mediator.Mediator`

Handles the connection between sasoptpy and the SAS Viya (CAS) server

Parameters

caller [*sasoptpy.Model* or *sasoptpy.Workspace*] Model or workspace that mediator belongs to

cas_session [`swat.cas.connection.CAS`] CAS connection

Notes

- CAS Mediator is used by *sasoptpy.Model* and *sasoptpy.Workspace* objects internally.

Model

CASMediator.solve(self, **kwargs)	Solve action for `Model` objects
CASMediator.tune(self, **kwargs)	Wrapper for the MILP tuner
CASMediator.tune_problem(self, **kwargs)	Calls optimization.tuner CAS action to finds out the ideal configuration
CASMediator.solve_with_mps(self, **kwargs)	Submits the problem in MPS (DataFrame) format, supported by old versions
CASMediator.solve_with_optmodel(self, **kwargs)	Submits the problem in OPTMODEL format
CASMediator.parse_cas_solution(self)	Performs post-solve operations
CASMediator.parse_cas_table(self, table)	Converts requested `swat.cas.table.CASTable` objects to `swat.dataframe.SASDataFrame`
CASMediator.set_variable_values(self, solution)	Performs post-solve assignment of variable values

Continued on next page

Table 28 – continued from previous page

CASMediator.set_constraint_values(self, solution)	Performs post-solve assignment of constraint values
CASMediator.set_model_objective_value(self)	Performs post-solve assignment of objective values
CASMediator.set_variable_init_values(self)	Performs post-solve assignment of variable initial values
CASMediator.upload_user_blocks(self)	Uploads user-defined decomposition blocks to the CAS server
CASMediator.upload_model(self[, name, . . .])	Converts internal model to MPS table and upload to CAS session

sasoptpy.interface.CASMediator.solve

`CASMediator.solve(self, **kwargs)`

Solve action for `Model` objects

sasoptpy.interface.CASMediator.tune

`CASMediator.tune(self, **kwargs)`

Wrapper for the MILP tuner

sasoptpy.interface.CASMediator.tune_problem

`CASMediator.tune_problem(self, **kwargs)`

Calls optimization.tuner CAS action to finds out the ideal configuration

sasoptpy.interface.CASMediator.solve_with_mps

`CASMediator.solve_with_mps(self, **kwargs)`

Submits the problem in MPS (DataFrame) format, supported by old versions

Parameters

kwargs [dict] Keyword arguments for solver settings and options

Returns

primal_solution [`swat.dataframe.SASDataFrame`] Solution of the model or None

sasoptpy.interface.CASMediator.solve_with_optmodel

`CASMediator.solve_with_optmodel(self, **kwargs)`

Submits the problem in OPTMODEL format

Parameters

kwargs [dict] Keyword arguments for solver settings and options

Returns

primal_solution [`swat.dataframe.SASDataFrame`] Solution of the model or None

sasoptpy.interface.CASMediator.parse_cas_solution

`CASMediator.parse_cas_solution(self)`

Performs post-solve operations

Returns

solution [`swat.dataframe.SASDataFrame`] Solution of the problem

sasoptpy.interface.CASMediator.parse_cas_table

`CASMediator.parse_cas_table(self, table)`

Converts requested `swat.cas.table.CASTable` objects to `swat.dataframe.SASDataFrame`

sasoptpy.interface.CASMediator.set_variable_values

`CASMediator.set_variable_values(self, solution)`

Performs post-solve assignment of variable values

Parameters

solution [class:*swat.dataframe.SASDataFrame*] Primal solution of the problem

sasoptpy.interface.CASMediator.set_constraint_values

`CASMediator.set_constraint_values(self, solution)`

Performs post-solve assignment of constraint values

Parameters

solution [class:*swat.dataframe.SASDataFrame*] Primal solution of the problem

sasoptpy.interface.CASMediator.set_model_objective_value

`CASMediator.set_model_objective_value(self)`

Performs post-solve assignment of objective values

Parameters

solution [class:*swat.dataframe.SASDataFrame*] Primal solution of the problem

sasoptpy.interface.CASMediator.set_variable_init_values

`CASMediator.set_variable_init_values(self)`

Performs post-solve assignment of variable initial values

Parameters

solution [class:*swat.dataframe.SASDataFrame*] Primal solution of the problem

sasoptpy.interface.CASMediator.upload_user_blocks

CASMediator.**upload_user_blocks**(*self*)

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}})
```

sasoptpy.interface.CASMediator.upload_model

CASMediator.**upload_model**(*self*, *name=None*, *replace=True*, *constant=False*, *verbose=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.

Workspace

`CASMediator.submit`(self, **kwargs)	Submit action for custom input and `sasoptpy.Workspace` objects
`CASMediator.submit_optmodel_code`(self, ...)	Converts caller into OPTMODEL code and submits using optimization.runOptmodel action
`CASMediator.parse_cas_workspace_response`(self)	Parses results of workspace submission
`CASMediator.set_workspace_variable_values`(...)	Performs post-solve assignment of `sasoptpy.Workspace` variable values

sasoptpy.interface.CASMediator.submit

CASMediator.**submit**(*self*, ***kwargs*)

Submit action for custom input and `sasoptpy.Workspace` objects

sasoptpy.interface.CASMediator.submit_optmodel_code

CASMediator.**submit_optmodel_code**(*self*, ***kwargs*)

Converts caller into OPTMODEL code and submits using optimization.runOptmodel action

Parameters

kwargs : Solver settings and options

sasoptpy.interface.CASMediator.parse_cas_workspace_response

CASMediator.**parse_cas_workspace_response**(*self*)

Parses results of workspace submission

sasoptpy.interface.CASMediator.set_workspace_variable_values

CASMediator.**set_workspace_variable_values**(*self*, *solution*)

Performs post-solve assignment of `sasoptpy.Workspace` variable values

SAS

`SASMediator`(caller, sas_session)	Handles the connection between sasoptpy and SAS instance

sasoptpy.interface.SASMediator

class SASMediator(*caller*, *sas_session*)

Bases: `sasoptpy.interface.solver.mediator.Mediator`

Handles the connection between sasoptpy and SAS instance

Parameters

caller [`sasoptpy.Model` or `sasoptpy.Workspace`] Model or workspace that mediator belongs to

sas_session [`saspy.SASsession`] SAS session object

Notes

- SASMediator is used by *sasoptpy.Model* and *sasoptpy.Workspace* objects internally.

Model

SASMediator.solve(self, **kwargs)	Solve action for `Model` objects
SASMediator.solve_with_mps(self, **kwargs)	Submits the problem in MPS (DataFrame) format, supported by old versions
SASMediator.solve_with_optmodel(self, **kwargs)	Submits the problem in OPTMODEL format
SASMediator.parse_sas_mps_solution(self)	Parses MPS solution after *solve* and returns solution
SASMediator.parse_sas_solution(self)	Performs post-solve operations
SASMediator.parse_sas_table(self, table_name)	Converts requested table name into `pandas.DataFrame`
SASMediator.convert_to_original(self, table)	Converts variable names to their original format if a placeholder gets used
SASMediator.perform_postsolve_operations(self)	Performs post-solve operations for proper output display

sasoptpy.interface.SASMediator.solve

`SASMediator.`**`solve`**(*self*, ***kwargs*)

Solve action for `Model` objects

sasoptpy.interface.SASMediator.solve_with_mps

`SASMediator.`**`solve_with_mps`**(*self*, ***kwargs*)

Submits the problem in MPS (DataFrame) format, supported by old versions

Parameters

kwargs [dict] Keyword arguments for solver settings and options

Returns

primal_solution [`pandas.DataFrame`] Solution of the model or None

sasoptpy.interface.SASMediator.solve_with_optmodel

`SASMediator.`**`solve_with_optmodel`**(*self*, ***kwargs*)

Submits the problem in OPTMODEL format

Parameters

kwargs [dict] Keyword arguments for solver settings and options

Returns

primal_solution [`pandas.DataFrame`] Solution of the model or None

sasoptpy.interface.SASMediator.parse_sas_mps_solution

SASMediator.**parse_sas_mps_solution**(*self*)

Parses MPS solution after *solve* and returns solution

sasoptpy.interface.SASMediator.parse_sas_solution

SASMediator.**parse_sas_solution**(*self*)

Performs post-solve operations

Returns

solution [`pandas.DataFrame`] Solution of the problem

sasoptpy.interface.SASMediator.parse_sas_table

SASMediator.**parse_sas_table**(*self*, *table_name*)

Converts requested table name into `pandas.DataFrame`

sasoptpy.interface.SASMediator.convert_to_original

SASMediator.**convert_to_original**(*self*, *table*)

Converts variable names to their original format if a placeholder gets used

sasoptpy.interface.SASMediator.perform_postsolve_operations

SASMediator.**perform_postsolve_operations**(*self*)

Performs post-solve operations for proper output display

Workspace

SASMediator.submit(self, **kwargs)	Submit action for custom input and *sasoptpy.Workspace* objects
SASMediator.submit_optmodel_code(self, . . .)	Submits given *sasoptpy.Workspace* object in OPTMODEL format
SASMediator.parse_sas_workspace_response(self)	Parses results of workspace submission
SASMediator.set_workspace_variable_values(. . .)	Performs post-solve assignment of *sasoptpy.Workspace* variable values

sasoptpy.interface.SASMediator.submit

SASMediator.**submit**(*self*, ***kwargs*)
Submit action for custom input and *sasoptpy.Workspace* objects

sasoptpy.interface.SASMediator.submit_optmodel_code

SASMediator.**submit_optmodel_code**(*self*, ***kwargs*)
Submits given *sasoptpy.Workspace* object in OPTMODEL format

Parameters

kwargs : Solver settings and options

sasoptpy.interface.SASMediator.parse_sas_workspace_response

SASMediator.**parse_sas_workspace_response**(*self*)
Parses results of workspace submission

sasoptpy.interface.SASMediator.set_workspace_variable_values

SASMediator.**set_workspace_variable_values**(*self*, *solution*)
Performs post-solve assignment of *sasoptpy.Workspace* variable values

5.4 Functions

5.4.1 Functions

Utility Functions

dict_to_frame(dictobj[, cols])	Converts dictionaries to DataFrame objects for pretty printing
exp_range(start, stop[, step])	Creates a set within specified range
flatten_frame(df[, swap])	Converts a pandas.DataFrame object into pandas.Series
get_value_table(*args, **kwargs)	Returns values of the given arguments as a merged pandas DataFrame
expr_sum(argv)	Summation function for *Expression* objects
quick_sum(argv)	Summation function for *Expression* objects
reset()	Resets package configs and internal counters

sasoptpy.dict_to_frame

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

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

Creates a set within specified 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.flatten_frame

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

sasoptpy.get_value_table

get_value_table (**args*, ***kwargs*)

Returns values of the given arguments as a merged pandas DataFrame

Parameters

key [list, optional] Keys for objects

rhs [bool, optional] Option for including constant values

Returns

table [pandas.DataFrame] DataFrame object that holds object values

sasoptpy.expr_sum

expr_sum(*argv*)
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.expr_sum(2*x[i] for i in range(10000))
```

sasoptpy.quick_sum

quick_sum(*argv*)
Summation function for *Expression* objects

Notes

This method will deprecate in future versions. Use *expr_sum()* instead.

sasoptpy.reset

reset()
Resets package configs and internal counters

Abstract Actions

actions.read_data(table, index, columns)	Reads data tables inside Set and Parameter objects
actions.create_data(table, index, columns)	Creates data tables from variables, parameters, and expressions
actions.solve([options, primalin])	Solves the active optimization problem and generates results
actions.for_loop(*args)	Creates a for-loop container to be executed on the server
actions.cofor_loop(*args)	Creates a cofor-loop to be executed on the server concurrently
actions.if_condition(logic_expression, ...)	Creates an if-else block
actions.switch_conditions(**args)	Creates several if-else blocks by using the specified arguments
actions.set_value(left, right)	Creates an assignment statement
actions.fix(*args)	Fixes values of variables to the specified values

Continued on next page

Table 34 – continued from previous page

actions.unfix(*args)	Unfixes values of variables
actions.set_objective(expression, name, sense)	Specifies the objective function
actions.print_item(*args)	Prints the specified argument list on server
actions.put_item(*args[, names])	Prints the specified item values to the output log
actions.expand()	Prints expanded problem to output
actions.drop(*args)	Drops the specified constraints or constraint groups from model
actions.restore(*args)	Restores dropped constraint and constraint groups
actions.union(*args)	Aggregates the specified sets and set expressions
actions.diff(left, right)	Gets the difference between set and set expressions
actions.substring(main_string, first_pos, . . .)	Gets the substring of the specified positions
actions.use_problem(problem)	Changes the currently active problem

sasoptpy.actions.read_data

read_data(*table*, *index*, *columns*)

Reads data tables inside Set and Parameter objects

Parameters

table [string or `swat.cas.table.CASTable`] Table object or name to be read, case-insensitive

index [dict] Index properties of the table

Has two main members:

- **target** [*sasoptpy.abstract.Set*] Target Set object to be read into
- **key** [string, list or None] Column name to be be read from.

  For multiple indices, key should be a list of string or *sasoptpy.abstract.SetIterator* objects

For a given set *YEAR* and column name *year_no*, the index dictionary should be written as:

```
>>> {'target': YEAR, 'key': 'year_no'}
```

If index is simply the row number, use *'key': so.N* which is equivalent to the special *_N_* character in the SAS language.

columns [list] A list of dictionaries, each holding column properties.

Columns are printed in the specified order. Each column should be represented as a dictionary with following fields:

- **target** [*sasoptpy.abstract.ParameterGroup*] Target parameter object to be read into
- **column** [string] Column name to be read from
- **index** [`sasoptpy.SetIterator`, optional] Subindex for specific column, needed for complex operations

If the name of the *sasoptpy.abstract.Parameter* object is same as the column name, the following call is enough:

```
>>> p = so.Parameter(name='price')
>>> read_data(..., columns=[{'target': p}])
```

For reading a different column name, *column* field should be specified:

```
>>> {'target': p, 'column': 'price_usd'}
```

When working with `ParameterGroup` objects, sometimes a secondary loop is needed. This is achieved by using the *index* field, along with the `sasoptpy.abstract.statement.ReadDataStatement.append()` method.

Returns

r [*sasoptpy.abstract.statement.ReadDataStatement*] Read data statement object, which includes all properties

Additional columns can be added using the `sasoptpy.abstract.statement.ReadDataStatement.append()` function.

See also:

tests.abstract.statement.test_read_data.TestReadData

Examples

Reading a regular set:

```
>>> with Workspace('test_workspace') as ws:
>>>     ITEMS = Set(name='ITEMS')
>>>     value = ParameterGroup(ITEMS, name='value', init=0)
>>>     get = VariableGroup(ITEMS, name='get', vartype=so.INT, lb=0)
>>>     read_data(
...         table="values",
...         index={'target': ITEMS, 'key': None},
...         columns=[{'target': value}])
>>> print(so.to_optmodel(w))
proc optmodel;
    set ITEMS;
    num value {ITEMS} init 0;
    var get {{ITEMS}} integer >= 0;
    read data values into ITEMS value;
quit;
```

Reading with row index:

```
>>> with so.Workspace('test_read_data_n') as ws:
>>>     ASSETS = so.Set(name='ASSETS')
>>>     ret = so.ParameterGroup(ASSETS, name='return', ptype=so.NUM)
>>>     read_data(
...         table='means',
...         index={'target': ASSETS, 'key': so.N},
...         columns=[{'target': ret}]
...     )
>>> print(so.to_optmodel(w))
proc optmodel;
    set ASSETS;
    num return {ASSETS};
```

(continues on next page)

(continued from previous page)

```
    read data means into ASSETS=[_N_] return;
quit;
```

Reading with no index set and subindex:

```
>>> with so.Workspace('test_read_data_no_index_expression') as ws:
>>>     ASSETS = so.Set(name='ASSETS')
>>>     cov = so.ParameterGroup(ASSETS, ASSETS, name='cov', init=0)
>>>     with iterate(ASSETS, 'asset1') as asset1, iterate(ASSETS, 'asset2') as␣
↪asset2:
>>>         read_data(
...             table='covdata',
...             index={'key': [asset1, asset2]},
...             columns=[
...                 {'target': cov},
...                 {'target': cov[asset2, asset1],
...                 'column': 'cov'}])
>>> print(so.to_optmodel(w))
proc optmodel;
    set ASSETS;
    num cov {ASSETS, ASSETS} init 0;
    read data covdata into [asset1 asset2] cov cov[asset2, asset1]=cov;
quit;
```

Reading a column with multiple indices:

```
>>> with so.Workspace(name='test_read_data_idx_col') as ws:
>>>     dow = so.Set(name='DOW', value=so.exp_range(1, 6))
>>>     locs = so.Set(name='LOCS', settype=so.STR)
>>>     demand = so.ParameterGroup(locs, dow, name='demand')
>>>     with iterate(locs, name='loc') as loc:
>>>         r = read_data(
...             table='dmnd',
...             index={'target': locs, 'key': loc}
...         )
>>>         with iterate(dow, name='d') as d:
>>>             r.append({
...                 'index': d,
...                 'target': demand[loc, d],
...                 'column': concat('day', d)
...             })
>>> optmodel_code = so.to_optmodel(ws)
proc optmodel;
    set DOW = 1..5;
    set <str> LOCS;
    num demand {LOCS, DOW};
    read data dmnd into LOCS=[loc] {d in DOW} < demand[loc, d]=col('day' || d) >;
quit;
```

sasoptpy.actions.create_data

create_data (*table*, *index*, *columns*)

Creates data tables from variables, parameters, and expressions

Parameters

table [string] Name of the table to be created

index [dict] Table index properties

This dictionary can be empty if no index is needed. It can have following fields:

- **key** [list] List of index keys. Keys can be string or *sasoptpy.abstract.SetIterator* objects
- **set** [list] List of sets that is being assigned to keys

columns [list] List of columns. Columns can be *sasoptpy.abstract.Parameter*, *sasoptpy.abstract.ParameterGroup* objects or dictionaries. If specified as a dictionary, each can have following keys:

- **name** [string] Name of the column in output table
- **expression** [sasoptpy.core.Expression] Any expression
- **index** [list or *sasoptpy.abstract.SetIterator*] Index for internal loops

The *index* field can be used when a subindex is needed. When specified as a list, members should be *sasoptpy.abstract.SetIterator* objects. See examples for more details.

See also:

tests.abstract.statement.test_create_data.TestCreateData

Examples

Regular column:

```
>>> with so.Workspace('w') as w:
>>>     m = so.Parameter(name='m', value=7)
>>>     n = so.Parameter(name='n', value=5)
>>>     create_data(table='example', index={}, columns=[m, n])
>>> print(so.to_optmodel(w))
proc optmodel;
    num m = 7;
    num n = 5;
    create data example from m n;
quit;
```

Column with name:

```
>>> with so.Workspace('w') as w:
>>>     m = so.Parameter(name='m', value=7)
>>>     n = so.Parameter(name='n', value=5)
>>>     create_data(table='example', index={}, columns=[
...         {'name': 'ratio', 'expression': m/n}
...     ])
>>> print(so.to_optmodel(w))
```

(continues on next page)

(continued from previous page)

```
proc optmodel;
    num m = 7;
    num n = 5;
    create data example from ratio=((m) / (n));
quit;
```

Column name with concat:

```
>>> from sasoptpy.util import concat
>>> with so.Workspace('w') as w:
>>>     m = so.Parameter(name='m', value=7)
>>>     n = so.Parameter(name='n', value=5)
>>>     create_data(table='example', index={}, columns=[
...         {'name': concat('s', n), 'expression': m+n}
...     ])
>>> print(so.to_optmodel(w))
proc optmodel;
    num m = 7;
    num n = 5;
    create data example from col('s' || n)=(m + n);
quit;
```

Table with index:

```
>>> with so.Workspace('w') as w:
>>>     m = so.ParameterGroup(
>>>         so.exp_range(1, 6), so.exp_range(1, 4), name='m', init=0)
>>>     m[1, 1] = 1
>>>     m[4, 1] = 1
>>>     S = so.Set(name='ISET', value=[i**2 for i in range(1, 3)])
>>>     create_data(
...         table='example',
...         index={'key': ['i', 'j'], 'set': [S, [1, 2]]},
...         columns=[m]
...     )
>>> print(so.to_optmodel(w))
proc optmodel;
    num m {1..5, 1..3} init 0;
    m[1, 1] = 1;
    m[4, 1] = 1;
    set ISET = {1,4};
    create data example from [i j] = {{ISET,{1,2}}} m;
quit;
```

Index over Python range:

```
>>> with so.Workspace('w') as w:
>>>     s = so.Set(name='S', value=so.exp_range(1, 6))
>>>     x = so.VariableGroup(s, name='x')
>>>     x[1] = 1
>>>     create_data(table='example',
...         index={'key': ['i'], 'set': so.exp_range(1, 4)}, columns=[x])
>>> print(so.to_optmodel(w))
proc optmodel;
    set S = 1..5;
    var x {{S}};
    x[1] = 1;
```

(continues on next page)

(continued from previous page)

```
    create data example from [i] = {1..3} x;
quit;
```

Append column with index:

```
>>> from sasoptpy.util import iterate, concat
>>> with so.Workspace('w', session=session) as w:
>>>     alph = so.Set(name='alph', settype=so.string, value=['a', 'b', 'c'])
>>>     x = so.VariableGroup([1, 2, 3], alph, name='x', init=2)
>>>     with iterate(so.exp_range(1, 4), name='i') as i:
>>>         c = create_data(
...             table='example',
...             index={'key': [i], 'set': [i.get_set()]})
>>>         with iterate(alph, name='j') as j:
>>>             c.append(
...                 {'name': concat('x', j),
...                  'expression': x[i, j],
...                  'index': j})
>>> print(so.to_optmodel(w))
proc optmodel;
    set <str> alph = {'a','b','c'};
    var x {{1,2,3}, {alph}} init 2;
    create data example from [i] = {{1..3}} {j in alph} < col('x' || j)=(x[i, j])␣
↪>;
quit;
```

Multiple column indices:

```
>>> from sasoptpy.util import concat, iterate
>>> with so.Workspace('w') as w:
>>>     S = so.Set(name='S', value=[1, 2, 3])
>>>     T = so.Set(name='T', value=[1, 3, 5])
>>>     x = so.VariableGroup(S, T, name='x', init=1)
>>>     with iterate(S, name='i') as i, iterate(T, name='j') as j:
>>>         create_data(
...             table='out',
...             index={},
...             columns=[
...                 {'name': concat('x', concat(i, j)), 'expression': x[i, j],
...                  'index': [i, j]}])
>>> print(so.to_optmodel(w))
proc optmodel;
    set S = {1,2,3};
    set T = {1,3,5};
    var x {{S}, {T}} init 1;
    create data out from {i in S, j in T} < col('x' || i || j)=(x[i, j]) >;
quit;
```

sasoptpy.actions.solve

solve (*options=None*, *primalin=False*)

Solves the active optimization problem and generates results

Parameters

options [dict, optional] Solver options

This dictionary can have several fields.

- **with** [string] Name of the solver, see possible values under Notes.

See *Solver Options* for a list of solver options. All fields in options (except *with*) is passed directly to the solver.

primalin [bool, optional] When set to *True*, uses existing variable values as an initial point in MILP solver

Returns

ss [`sasoptpy.abstract.statement.SolveStatement`] Solve statement object.

Contents of the response can be retrieved using *get_response* function.

Notes

Possible solver names for *with* parameter:

- *lp* : Linear programming
- *milp* : Mixed integer linear programming
- *nlp* : General nonlinear programming
- *qp* : Quadratic programming
- *blackbox* : Black-box optimization

SAS Optimization also has a constraint programming solver (clp), and network solver (network) but they are not currently supported by sasoptpy.

Examples

Regular solve:

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x', lb=1, ub=10)
>>>     o = so.Objective(2*x, sense=so.maximize, name='obj')
>>>     s = solve()
>>>     p = print_item(x)
>>> print(so.to_optmodel(w))
proc optmodel;
    var x >= 1 <= 10;
    max obj = 2 * x;
    solve;
    print x;
quit;
```

Option alternatives:

```
>>> with so.Workspace('w') as w:
>>>     # Problem declaration, etc..
>>>     solve()
>>>     solve(options={'with': 'milp'})
>>>     solve(options={'with': 'milp'}, primalin=True)
>>>     solve(options={'with': 'milp', 'presolver': None, 'feastol': 1e-6,
>>>                    'logfreq': 2, 'maxsols': 3, 'scale': 'automatic',
>>>                    'restarts': None, 'cutmir': 'aggressive'})
>>> print(so.to_optmodel(w))
proc optmodel;
    solve;
    solve with milp;
    solve with milp / primalin;
    solve with milp / presolver=None feastol=1e-06 logfreq=2 maxsols=3␣
↪scale=automatic restarts=None cutmir=aggressive;
quit;
```

sasoptpy.actions.for_loop

for_loop(**args*)

Creates a for-loop container to be executed on the server

Parameters

args [`sasoptpy.abstract.Set` objects] Any number of `sasoptpy.abstract.Set` objects can be given

Returns

set_iterator [`sasoptpy.abstract.SetIterator`, `sasoptpy.abstract.SetIteratorGroup`] Set iterators to be used inside for-loop

See also:

`sasoptpy.actions.cofor_loop()`

Notes

For tasks that can be run concurrently, consider using `sasoptpy.actions.cofor_loop()`

Examples

Regular for loop:

```
>>> with so.Workspace('w') as w:
>>>     r = so.exp_range(1, 11)
>>>     x = so.VariableGroup(r, name='x')
>>>     for i in for_loop(r):
>>>         x[i] = 1
>>> print(so.to_optmodel(w))
proc optmodel;
    var x {{1,2,3,4,5,6,7,8,9,10}};
    for {o13 in 1..10} do;
        x[o13] = 1;
```

(continues on next page)

(continued from previous page)

```
    end;
quit;
```

Nested for loops:

```
>>> from sasoptpy.actions import put_item
>>> with so.Workspace('w') as w:
>>>     for i in for_loop(range(1, 3)):
>>>         for j in for_loop(['a', 'b']):
>>>             put_item(i, j)
>>> print(so.to_optmodel(w))
proc optmodel;
    for {o2 in 1..2} do;
        for {o5 in {'a','b'}} do;
            put o2 o5;
        end;
    end;
quit;
```

Multiple set for-loops:

```
>>> with so.Workspace('w') as w:
>>>     r = so.Set(name='R', value=range(1, 11))
>>>     c = so.Set(name='C', value=range(1, 6))
>>>     a = so.ParameterGroup(r, c, name='A', ptype=so.number)
>>>     for (i, j) in for_loop(r, c):
>>>         a[i, j] = 1
>>> print(so.to_optmodel(w))
proc optmodel;
    set R = 1..10;
    set C = 1..5;
    num A {R, C};
    for {o5 in R, o7 in C} do;
        A[o5, o7] = 1;
    end;
quit;
```

sasoptpy.actions.cofor_loop

cofor_loop(**args*)

Creates a cofor-loop to be executed on the server concurrently

Parameters

args [`sasoptpy.abstract.Set` objects] Any number of `sasoptpy.abstract.Set` objects can be specified

Returns

set_iterator [`sasoptpy.abstract.SetIterator`, `sasoptpy.abstract.SetIteratorGroup`] Set iterators to be used inside cofor-loop

See also:

`sasoptpy.actions.for_loop()`

Notes

A cofor-loop runs its content concurrently. For tasks that depend on each other, consider using `sasoptpy.actions.for_loop()`

Examples

```
>>> with so.Workspace('w') as w:
>>>     x = so.VariableGroup(6, name='x', lb=0)
>>>     so.Objective(
>>>         so.expr_sum(x[i] for i in range(6)), name='z', sense=so.MIN)
>>>     a1 = so.Constraint(x[1] + x[2] + x[3] <= 4, name='a1')
>>>     for i in cofor_loop(so.exp_range(3, 6)):
>>>         fix(x[1], i)
>>>         solve()
>>>         put_item(i, x[1], so.Symbol('_solution_status_'), names=True)
>>> print(so.to_optmodel(w))
proc optmodel;
    var x {{0,1,2,3,4,5}} >= 0;
    min z = x[0] + x[1] + x[2] + x[3] + x[4] + x[5];
    con a1 : x[1] + x[2] + x[3] <= 4;
    cofor {o13 in 3..5} do;
        fix x[1]=o13;
        solve;
        put o13= x[1]= _solution_status_=;
    end;
quit;
```

sasoptpy.actions.if_condition

if_condition(*logic_expression*, *if_statement*, *else_statement=None*)

Creates an if-else block

Parameters

logic_expression [`sasoptpy.Constraint` or `sasoptpy.abstract.Condition`] Logical condition for the True case

For the condition, it is possible to combine constraints, such as

```
>>> a = so.Parameter(value=5)
>>> if_condition((a < 3) | (a > 6), func1, func2)
```

Constraints should be combined using bitwise operators (& for *and*, | for *or*).

if_statement [function or `IfElseStatement`] Python function or if-else statement to be called if the condition is True

else_statement [function or `IfElseStatement`, optional] Python function or if-else statement to be called if the condition is False

Examples

Regular condition:

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x')
>>>     x.set_value(0.5)
>>>     def func1():
>>>         x.set_value(1)
>>>     def func2():
>>>         x.set_value(0)
>>>     if_condition(x > 1e-6, func1, func2)
>>> print(so.to_optmodel(w))
proc optmodel;
    var x;
    x = 0.5;
    if x > 1e-06 then do;
        x = 1;
    end;
    else do;
        x = 0;
    end;
quit;
```

Combined conditions:

```
>>> with so.Workspace('w') as w:
>>>     p = so.Parameter(name='p')
>>>     def case1():
>>>         p.set_value(10)
>>>     def case2():
>>>         p.set_value(20)
>>>     r = so.Parameter(name='r', value=10)
>>>     if_condition((r < 5) | (r > 10), case1, case2)
>>> print(so.to_optmodel(w))
proc optmodel;
    num p;
    num r = 10;
    if (r < 5) or (r > 10) then do;
        p = 10;
    end;
    else do;
        p = 20;
    end;
quit;
```

sasoptpy.actions.switch_conditions

switch_conditions(***args*)

Creates several if-else blocks by using the specified arguments

Parameters

args : Several arguments can be passed to the function

Each case should follow a condition. You can use `sasoptpy.Constraint` objects as conditions, and Python functions for the cases.

Examples

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x')
>>>     p = so.Parameter(name='p')
>>>     x.set_value(2.5)
>>>     def func1():
>>>         p.set_value(1)
>>>     def func2():
>>>         p.set_value(2)
>>>     def func3():
>>>         p.set_value(3)
>>>     def func4():
>>>         p.set_value(0)
>>>     switch_conditions(x < 1, func1, x < 2, func2, x < 3, func3, func4)
>>> print(to.optmodel(w))
proc optmodel;
    var x;
    num p;
    x = 2.5;
    if x < 1 then do;
        p = 1;
    end;
    else if x < 2 then do;
        p = 2;
    end;
    else if x < 3 then do;
        p = 3;
    end;
    else do;
        p = 0;
    end;
quit;
```

sasoptpy.actions.set_value

set_value (*left, right*)

Creates an assignment statement

Parameters

left [*sasoptpy.Expression*] Any expression (variable or parameter)

right [*sasoptpy.Expression* or float] Right-hand-side expression

Examples

```
>>> with so.Workspace('ex_9_1_matirx_sqrt', session=None) as w:
>>>     so.LiteralStatement('call streaminit(1);')
>>>     n = so.Parameter(name='n', value=5)
>>>     rn = so.Set(name='RN', value=so.exp_range(1, n))
>>>     A = so.ParameterGroup(rn, rn, name='A', value="10-20*rand('UNIFORM')")
>>>     P = so.ParameterGroup(rn, rn, name='P')
>>>     for i in for_loop(rn):
>>>         for j in for_loop(so.exp_range(i, n)):
```

(continues on next page)

(continued from previous page)

```
>>>             set_value(P[i, j], so.expr_sum(A[i, k] * A[j, k] for k in rn))
>>> print(so.to_optmodel(w))
proc optmodel;
    call streaminit(1);
    num n = 5;
    set RN = 1..n;
    num A {RN, RN} = 10-20*rand('UNIFORM');
    num P {RN, RN};
    for {o7 in RN} do;
        for {o10 in o7..n} do;
            P[o7, o10] = sum {k in RN} (A[o7, k] * A[o10, k]);
        end;
    end;
quit;
```

sasoptpy.actions.fix

fix (**args*)

Fixes values of variables to the specified values

Parameters

args [*sasoptpy.Variable*, float, *sasoptpy.Expression*, tuple] Set of arguments to be fixed

Arguments get paired (if not given in tuples) to allow several fix operations

See also:

sasoptpy.actions.unfix()

tests.abstract.statement.test_fix_unfix.TestFix

Examples

Regular fix statement:

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x')
>>>     fix(x, 1)
>>>     solve()
>>>     unfix(x)
>>> print(so.to_optmodel(w))
proc optmodel;
    var x;
    fix x=1;
    solve;
    unfix x;
quit;
```

Multiple fix-unfix:

```
>>> with so.Workspace('w') as w:
>>>     x = so.VariableGroup(4, name='x')
>>>     for i in cofor_loop(range(4)):
```

(continues on next page)

(continued from previous page)

```
>>>         fix((x[0], i), (x[1], 1))
>>>         solve()
>>>         unfix(x[0], (x[1], 2))
>>> print(so.to_optmodel(w))
proc optmodel;
    var x {{0,1,2,3}};
    cofor {o7 in 0..3} do;
        fix x[0]=o7 x[1]=1;
        solve;
        unfix x[0] x[1]=2;
    end;
quit;
```

sasoptpy.actions.unfix

unfix(**args*)

Unfixes values of variables

Parameters

args [*sasoptpy.Variable* objects] Set of arguments to be unfixed

See also:

sasoptpy.actions.fix()

tests.abstract.statement.test_fix_unfix.TestFix

Examples

Regular unfix statement:

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x')
>>>     fix(x, 1)
>>>     solve()
>>>     unfix(x)
>>> print(so.to_optmodel(w))
proc optmodel;
    var x;
    fix x=1;
    solve;
    unfix x;
quit;
```

Multiple fix-unfix:

```
>>> with so.Workspace('w') as w:
>>>     x = so.VariableGroup(4, name='x')
>>>     for i in cofor_loop(range(4)):
>>>         fix((x[0], i), (x[1], 1))
>>>         solve()
>>>         unfix(x[0], (x[1], 2))
>>> print(so.to_optmodel(w))
proc optmodel;
```

(continues on next page)

(continued from previous page)

```
    var x {{0,1,2,3}};
    cofor {o7 in 0..3} do;
        fix x[0]=o7 x[1]=1;
        solve;
        unfix x[0] x[1]=2;
    end;
quit;
```

sasoptpy.actions.set_objective

set_objective(*expression*, *name*, *sense*)
Specifies the objective function

Parameters

expression [*sasoptpy.Expression*] Objective function

name [string] Name of the objective function

sense [string] Direction of the objective function, *so.MAX* or *so.MIN*

Examples

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x', lb=1)
>>>     set_objective(x ** 3, name='xcube', sense=so.minimize)
>>>     solve()
>>> print(so.to_optmodel(w))
proc optmodel;
    var x >= 1;
    MIN xcube = (x) ^ (3);
    solve;
quit;
```

sasoptpy.actions.print_item

print_item(**args*)
Prints the specified argument list on server

Parameters

args [*sasoptpy.Variable*, *sasoptpy.Expression*] Arbitrary number of arguments to be printed

These values are printed on the server, but can be retrieved after execution

Returns

ps [*sasoptpy.abstract.statement.PrintStatement*] Print statement object.

Contents of the response can be retrieved using *get_response* function.

Examples

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x', lb=1, ub=10)
>>>     o = so.Objective(2*x, sense=so.maximize, name='obj')
>>>     s = solve()
>>>     p = print_item(x)
>>> print(so.to_optmodel(w))
proc optmodel;
    var x >= 1 <= 10;
    max obj = 2 * x;
    solve;
    print x;
quit;
>>> print(p.get_response())
      x
0  10.0
```

sasoptpy.actions.put_item

put_item (**args*, *names=None*)

Prints the specified item values to the output log

Parameters

args [*sasoptpy.Expression*, string] Arbitrary elements to be put into log

Variables, variable groups, and expressions can be printed to log

names [bool, optional] When set to *True*, prints the name of the arguments in the log

Examples

Regular operation:

```
>>> with so.Workspace('w') as w:
>>>     for i in for_loop(range(1, 3)):
>>>         for j in for_loop(['a', 'b']):
>>>             put_item(i, j)
>>> print(so.to_optmodel(w))
proc optmodel;
    for {o2 in 1..2} do;
        for {o5 in {'a','b'}} do;
            put o2 o5;
        end;
    end;
quit;
```

Print with names:

```
>>> with so.Workspace('w') as w:
>>>     x = so.VariableGroup(6, name='x', lb=0)
>>>     so.Objective(
>>>         so.expr_sum(x[i] for i in range(6)), name='z', sense=so.MIN)
>>>     a1 = so.Constraint(x[1] + x[2] + x[3] <= 4, name='a1')
>>>     for i in cofor_loop(so.exp_range(3, 6)):
```

(continues on next page)

(continued from previous page)

```
>>>         fix(x[1], i)
>>>         solve()
>>>         put_item(i, x[1], so.Symbol('_solution_status_'), names=True)
proc optmodel;
    var x {{0,1,2,3,4,5}} >= 0;
    min z = x[0] + x[1] + x[2] + x[3] + x[4] + x[5];
    con a1 : x[1] + x[2] + x[3] <= 4;
    cofor {o13 in 3..5} do;
        fix x[1]=o13;
        solve;
        put o13= x[1]= _solution_status_=;
    end;
quit;
```

sasoptpy.actions.expand

expand()

Prints expanded problem to output

Examples

```
>>> with so.Workspace(name='w') as w:
>>>     x = so.VariableGroup(3, name='x')
>>>     self.assertEqual(x[0].sym.get_conditions_str(), '')
>>>     # solve
>>>     x[0].set_value(1)
>>>     x[1].set_value(5)
>>>     x[2].set_value(0)
>>>     c = so.ConstraintGroup(None, name='c')
>>>     with iterate([0, 1, 2], 's') as i:
>>>         with condition(x[i].sym > 0):
>>>             c[i] = x[i] >= 1
>>>     set_objective(x[0], name='obj', sense=so.MIN)
>>>     expand()
>>>     solve()
>>> print(so.to_optmodel(w))
proc optmodel;
    var x {{0,1,2}};
    x[0] = 1;
    x[1] = 5;
    x[2] = 0;
    con c {s in {0,1,2}: x[s].sol > 0} : x[s] >= 1;
    MIN obj = x[0];
    expand;
    solve;
quit;
```

sasoptpy.actions.drop

drop(**args*)

Drops the specified constraints or constraint groups from model

Parameters

args [*sasoptpy.Constraint*, *sasoptpy.ConstraintGroup*] Constraints to be dropped

See also:

sasoptpy.actions.restore()

Examples

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x', lb=1)
>>>     y = so.Variable(name='y', lb=0)
>>>     c = so.Constraint(sm.sqrt(x) >= 5, name='c')
>>>     o = so.Objective(x + y, sense=so.MIN, name='obj')
>>>     s = solve()
>>>     drop(c)
>>>     o2 = so.Objective(x, sense=so.MIN, name='obj2')
>>>     s2 = solve()
>>> print(so.to_optmodel(w))
proc optmodel;
    var x >= 1;
    var y >= 0;
    con c : sqrt(x) >= 5;
    min obj = x + y;
    solve;
    drop c;
    min obj2 = x;
    solve;
quit;
```

sasoptpy.actions.restore

restore(**args*)

Restores dropped constraint and constraint groups

Parameters

args [*sasoptpy.Constraint*, *sasoptpy.ConstraintGroup*] Constraints to be restored

See also:

sasoptpy.actions.drop()

Examples

```
>>> with so.Workspace('w') as w:
>>>     x = so.Variable(name='x', lb=-1)
>>>     set_objective(x**3, name='xcube', sense=so.minimize)
>>>     c = so.Constraint(x >= 1, name='xbound')
>>>     solve()
>>>     drop(c)
>>>     solve()
>>>     restore(c)
>>>     solve()
>>> print(so.to_optmodel(w))
proc optmodel;
    var x >= -1;
    MIN xcube = (x) ^ (3);
    con xbound : x >= 1;
    solve;
    drop xbound;
    solve;
    restore xbound;
    solve;
quit;
```

sasoptpy.actions.union

union (**args*)

Aggregates the specified sets and set expressions

Parameters

args [`sasoptpy.abstract.Set` and `sasoptpy.abstract.InlineSet`] Objects to be aggregated

Examples

```
>>> from sasoptpy.actions import union, put_item
>>> with so.Workspace('w') as w:
>>>     n = so.Parameter(name='n', value=11)
>>>     S = so.Set(name='S', value=so.exp_range(1, n))
>>>     T = so.Set(name='T', value=so.exp_range(n+1, 20))
>>>     U = so.Set(name='U', value=union(S, T))
>>>     put_item(U, names=True)
>>> print(so.to_optmodel(w))
proc optmodel;
    num n = 11;
    set S = 1..n;
    set T = n+1..20;
    set U = S union T;
    put U=;
quit;
```

sasoptpy.actions.diff

diff (*left*, *right*)

Gets the difference between set and set expressions

Parameters

left [*sasoptpy.abstract.Set*] Left operand

right [*sasoptpy.abstract.Set*] Right operand

Examples

```
>>> from sasoptpy.actions import diff, put_item
>>> with so.Workspace('w') as w:
>>>     S = so.Set(name='S', value=so.exp_range(1, 20))
>>>     T = so.Set(name='T', value=so.exp_range(1, 15))
>>>     U = so.Set(name='U', value=diff(S, T))
>>>     put_item(U, names=True)
>>> print(so.to_optmodel(w))
proc optmodel;
    set S = 1..19;
    set T = 1..14;
    set U = S diff T;
    put U=;
quit;
```

sasoptpy.actions.substring

substring (*main_string*, *first_pos*, *last_pos*)

Gets the substring of the specified positions

Parameters

main_string [*sasoptpy.abstract.Parameter* or string] Main string

first_pos [integer] First position of the substring, starting from 1

last_pos [integer] Last position of the substring

Examples

```
>>> with so.Workspace('w') as w:
>>>     p = so.Parameter(name='p', value='random_string', ptype=so.STR)
>>>     r = so.Parameter(name='r', value=substring(p, 1, 6), ptype=so.STR)
>>>     put_item(r)
>>> print(so.to_optmodel(w))
proc optmodel;
    str p = 'random_string';
    str r = substr(p, 1, 6);
    put r;
quit;
```

sasoptpy.actions.use_problem

use_problem(*problem*)

Changes the currently active problem

Parameters

problem [*sasoptpy.Model*] Model to be activated

Examples

```
>>> from sasoptpy.actions import use_problem
>>> with so.Workspace('w') as w:
>>>     m = so.Model(name='m')
>>>     m2 = so.Model(name='m2')
>>>     use_problem(m)
>>>     x = so.Variable(name='x')
>>>     use_problem(m2)
>>>     m.solve()
>>>     m2.solve()
>>> print(so.to_optmodel(w))
proc optmodel;
    problem m;
    problem m2;
    use problem m;
    var x;
    use problem m2;
    use problem m;
    solve;
    use problem m2;
    solve;
quit;
```

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 in base 2
math.log10(exp)	Logarithm function in 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

math_func (*exp*, *op*, **args*)
Function wrapper for math functions

Parameters

exp [`Expression`] Expression where the math function will be applied

op [string] String representation of the math function

args [float, optional] Additional arguments

sasoptpy.math.abs

abs (*exp*)
Absolute value function

sasoptpy.math.log

log (*exp*)
Natural logarithm function

sasoptpy.math.log2

log2 (*exp*)
Logarithm function in base 2

sasoptpy.math.log10

log10 (*exp*)
Logarithm function in base 10

sasoptpy.math.exp

exp (*exp*)
Exponential function

sasoptpy.math.sqrt

sqrt (*exp*)
Square root function

sasoptpy.math.mod

mod(*exp*, *divisor*)
Modulo function

Parameters

exp [Expression] Dividend

divisor [Expression] Divisor

sasoptpy.math.int

int(*exp*)
Integer value function

sasoptpy.math.sign

sign(*exp*)
Sign value function

sasoptpy.math.max

max(*exp*, **args*)
Largest value function

sasoptpy.math.min

min(*exp*, **args*)
Smallest value function

sasoptpy.math.sin

sin(*exp*)
Sine function

sasoptpy.math.cos

cos(*exp*)
Cosine function

sasoptpy.math.tan

tan (*exp*)
Tangent function

5.5 Tests

5.5.1 Unit Tests

Core

test_expression. TestExpression([methodName])	Unit tests for *sasoptpy.Expression* objects
test_objective.TestObjective([methodName])	Unit tests for *sasoptpy.Objective* objects
test_model.TestModel([methodName])	Unit tests for *sasoptpy.Model* objects
test_variable.TestVariable([methodName])	Unit tests for *sasoptpy.Variable* objects
test_variable_group. TestVariableGroup([. . .])	Unit tests for *sasoptpy.VariableGroup* objects
test_constraint. TestConstraint([methodName])	Unit tests for *sasoptpy.Constraint* objects
test_constraint_group. TestConstraintGroup([. . .])	Unit tests for *sasoptpy.ConstraintGroup* objects
test_util.TestUtil([methodName])	Unit tests for core utility functions

tests.core.test_expression.TestExpression

class TestExpression (*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Unit tests for *sasoptpy.Expression* objects

tests.core.test_objective.TestObjective

class TestObjective (*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Unit tests for *sasoptpy.Objective* objects

tests.core.test_model.TestModel

class TestModel (*methodName='runTest'*)
Bases: `unittest.case.TestCase`

Unit tests for *sasoptpy.Model* objects

tests.core.test_variable.TestVariable

class TestVariable(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.Variable` objects

tests.core.test_variable_group.TestVariableGroup

class TestVariableGroup(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.VariableGroup` objects

tests.core.test_constraint.TestConstraint

class TestConstraint(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.Constraint` objects

tests.core.test_constraint_group.TestConstraintGroup

class TestConstraintGroup(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.ConstraintGroup` objects

tests.core.test_util.TestUtil

class TestUtil(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for core utility functions

Abstract

`test_math.TestAbstractMath`([methodName])	Unit tests for mathematical functions
`test_set.TestSet`([methodName])	Unit tests for `sasoptpy.abstract.Set` objects
`test_set_iterator.TestSetIterator`([methodName])	Unit tests for `sasoptpy.abstract.SetIterator` objects
`test_parameter.TestParameter`([methodName])	
`test_implicit_variable.TestImplicitVariable`([. . .])	Unit tests for `sasoptpy.abstract.ImplicitVar` objects
`test_condition.TestCondition`([methodName])	Unit tests for abstract conditions to be executed on the server
`statement.test_assignment.TestAssignment`([. . .])	Unit tests for assignment statements

Continued on next page

Table 37 – continued from previous page

statement.test_cofor_loop.TestCoforLoop([…])	Unit tests for concurrent for (COFOR) statements
statement.test_create_data.TestCreateData([…])	Unit tests for CREATE DATA statements
statement.test_drop_restore.TestDropRestore([…])	Unit tests for DROP and RESTORE statements
statement.test_fix_unfix.TestFix([methodName])	Unit tests for FIX and UNFIX statements
statement.test_for_loop.TestForLoop([methodName])	Unit tests for FOR statements
statement.test_if_else.TestIfElse([methodName])	Unit tests for IF/ELSE and SWITCH statements
statement.test_literal.TestLiteral([methodName])	Unit tests for literal statements
statement.test_read_data.TestReadData([…])	Unit tests for READ DATA statements
statement.test_solve.TestSolve([methodName])	Unit tests for SOLVE statements

tests.abstract.test_math.TestAbstractMath

class TestAbstractMath(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for mathematical functions

tests.abstract.test_set.TestSet

class TestSet(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.abstract.Set` objects

tests.abstract.test_set_iterator.TestSetIterator

class TestSetIterator(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.abstract.SetIterator` objects

tests.abstract.test_parameter.TestParameter

class TestParameter(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

tests.abstract.test_implicit_variable.TestImplicitVariable

class TestImplicitVariable(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for `sasoptpy.abstract.ImplicitVar` objects

tests.abstract.test_condition.TestCondition

class TestCondition(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for abstract conditions to be executed on the server

tests.abstract.statement.test_assignment.TestAssignment

class TestAssignment(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for assignment statements

tests.abstract.statement.test_cofor_loop.TestCoforLoop

class TestCoforLoop(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for concurrent for (COFOR) statements

tests.abstract.statement.test_create_data.TestCreateData

class TestCreateData(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for CREATE DATA statements

tests.abstract.statement.test_drop_restore.TestDropRestore

class TestDropRestore(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for DROP and RESTORE statements

tests.abstract.statement.test_fix_unfix.TestFix

class TestFix(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for FIX and UNFIX statements

tests.abstract.statement.test_for_loop.TestForLoop

class TestForLoop(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for FOR statements

tests.abstract.statement.test_if_else.TestIfElse

class TestIfElse(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for IF/ELSE and SWITCH statements

tests.abstract.statement.test_literal.TestLiteral

class TestLiteral(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for literal statements

tests.abstract.statement.test_read_data.TestReadData

class TestReadData(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for READ DATA statements

tests.abstract.statement.test_solve.TestSolve

class TestSolve(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for SOLVE statements

Interface

`test_cas_interface.TestCASInterface`([methodName])	Unit tests for the CAS interface
`test_sas_interface.TestSASInterface`([methodName])	Unit tests for the SAS interface

tests.interface.test_cas_interface.TestCASInterface

class TestCASInterface(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for the CAS interface

tests.interface.test_sas_interface.TestSASInterface

class TestSASInterface(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for the SAS interface

Session

`test_workspace.TestWorkspace`([methodName])	Unit tests for the `sasoptpy.Workspace` objects

tests.session.test_workspace.TestWorkspace

class TestWorkspace(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

Unit tests for the `sasoptpy.Workspace` objects

CHAPTER

SIX

VERSION HISTORY

This page outlines changes from each release.

6.1 v0.2.1 (February 26, 2019)

6.1.1 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

6.1.2 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 by using the *yield* keyword for performance
- *key_name* and *col_names* parameters are added into `read_table()`

6.1.3 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

6.1.4 Notes

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

6.2 v0.2.0 (July 30, 2018)

6.2.1 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 code 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 returned when using *solveMilp* action and can be retrieved by using `Model.get_solution()` method
- `Model.get_variable_value()` is added to get solution values of abstract variables

6.2.2 Changes

- Variable and constraint naming schemes are replaced with OPTMODEL equivalent versions
- Variables and constraints now preserve the order in which they are inserted to the problem
- `Model.to_frame()` method is updated to reflect changes to VariableGroup and ConstraintGroup orderings
- Two solve methods, *solve_on_cas* and *solve_on_viya* are merged into `Model.solve()`
- `Model.solve()` method checks the available CAS actions and uses the runOptmodel action 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 use of MPS mode and *solveLp-solveMilp* actions
- Minor changes are applied to *__str__* and *__repr__* methods
- Creation indices for objects are being kept by using the return of the `register_name()` function

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

6.2.3 Bug Fixes

- Nondeterministic behavior when generating MPS files is fixed.

6.2.4 Notes

- Abstract and nonlinear models can be solved on Viya only if the 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.

6.3 v0.1.2 (April 24, 2018)

6.3.1 New Features

- As an experimental feature, sasoptpy now supports *SASPy* connections
- `Model.solve_local()` method is added for solving optimization problems by 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 retrieve `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

6.3.2 Changes

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

6.3.3 Bug Fixes

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

6.3.4 Notes

- A .gitignore file is added to the repository.
- A new example is added: Decentralization. Both *CAS* and *SAS* versions of the new example are added.
- 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.

6.4 v0.1.1 (February 26, 2018)

6.4.1 New Features

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

6.4.2 Changes

- methods.py is renamed to utils.py

6.4.3 Bug Fixes

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

6.5 v0.1.0 (December 22, 2017)

- Initial release

PYTHON MODULE INDEX

S

INDEX

Symbols

A

C

D

E

F

G

S

T

U

V

W