Paper SAS1814-2018

Optimization Modeling with Python and SAS® Viya®

Jared Erickson and Sertalp B. Cay, SAS Institute Inc.

ABSTRACT

Python has become a popular programming language for both data analytics and mathematical optimization. With SAS® Viya® and its Python interface, Python programmers can use the state-of-the-art optimization solvers that SAS® provides. This paper demonstrates an approach for Python programmers to naturally model their optimization problems, solve them by using SAS® Optimization solver actions, and view and interact with the results. The common tools for using the optimization solvers in SAS for these purposes are the OPTMODEL and IML procedures, but programmers more familiar with Python might find this alternative approach easier to grasp.

INTRODUCTION

Algebraic modeling languages (AMLs) enable modelers to formulate optimization problems in a natural way that can conveniently be solved by optimization solvers. Direct input for solvers, such as unlabeled lists of coefficients, is generally not easily readable by humans. An AML lets users give constraints and variables names that make sense to them, so it is easy to modify the model or build it in pieces. The OPTMODEL procedure is the AML available in SAS/OR® software. PROC OPTMODEL is a very powerful tool that you can use to model a wide variety of mathematical optimization problem types, including linear programming (LP), mixed integer linear programming (MILP), quadratic programming (QP), constraint logic programming (CLP), nonlinear programming (NLP), and network analysis.

Although PROC OPTMODEL is easy and intuitive for SAS users, many people who work on analytics and operations research problems are more comfortable using the Python language. With SAS Viya, Python users can access some of the SAS Optimization solvers in the **`optimization`** action set on the Cloud Analytic Services (CAS) server. The CAS server holds data and can run analytics actions on the data. (For more information about CAS, see *SAS Viya: System Programming Guide*.) The **`optimization`** action set includes actions for the LP, MILP, and QP solvers. The purpose of this paper is to introduce the *sasoptpy* modeling package, which gives you the ability to write optimization models in Python that it can solve using SAS Optimization solvers. This package provides a convenient modeling framework for programmers who already use Python. These users can also benefit from integrating optimization models into their Python code by using native Python functionality. The *sasoptpy* package is platform-independent, so you can use it on Windows, Linux, and macOS operating systems.

You can find the full documentation with examples and installation instructions at https://sassoftware.github.io/sasoptpy/. The *sasoptpy* package is an open-source Python package, and you can find the source code at https://github.com/sassoftware/sasoptpy under Apache 2.0 License. This paper is based on version 0.1.1.

OVERVIEW OF MATHEMATICAL OPTIMIZATION

Mathematical optimization, also known as mathematical programming, is a way of modeling and solving problems that require choosing the best option from a feasible set. The major components of an optimization model include the following:

- Decision variables: Variables whose values are determined by the solution algorithm
- Objective: A function whose value is to be optimized
- Constraints: Equalities and inequalities that restrict the set of feasible solutions

Mathematical optimization is used in a wide variety of industries, including finance, health care, and marketing. Applications include logistics, production planning, scheduling, location, and portfolio optimization problems. For more information and examples, see *SAS/OR User's Guide: Mathematical Programming* and *SAS Optimization: Mathematical Optimization Procedures*.

FIRST EXAMPLE: CANDY MANUFACTURER PRODUCT-MIX PROBLEM

For a simple example, consider the model in the section "Model Building with PROC OPTMODEL" of *SAS/OR User's Guide: Mathematical Programming*. A candy manufacturer makes two products, chocolate and toffee. It uses four processes that have a limited amount of time available per day (7.5 hours, or 27,000 seconds), and the production of one pound of each product requires different amounts of time in each process. The manufacturer makes a profit of \$0.25 per pound of chocolate and \$0.75 per pound of toffee, and it would like to maximize its total profit. You can write the mathematical representation of the optimization problem as follows:

$$
\begin{array}{lrcl}
\text{maximize} & 0.25 \cdot \text{chocolate} + 0.75 \cdot \text{toffee} & & \\
\text{subject to} & 15 \cdot \text{chocolate} + 40 \cdot \text{toffee} & \leq & 27000 \\
& 56.25 \cdot \text{toffee} & \leq & 27000 \\
& 18.75 \cdot \text{chocolate} & \leq & 27000 \\
& 12 \cdot \text{chocolate} + 50 \cdot \text{toffee} & \leq & 27000 \\
& \text{chocolate}, \text{toffee} & \geq & 0
\end{array}
$$

Here is the Python code to create the same model by using the *sasoptpy* package, which solves it with the CAS action **solveLp**, and the corresponding PROC OPTMODEL code:

```
from swat import CAS                                #
import sasoptpy as so                               #
s = CAS(hostname, port)                             #
m = so.Model(name='candy', session=s)               #  proc optmodel;
choco = m.add_variable(lb=0, name='choco')          #     var choco >= 0;
toffee = m.add_variable(lb=0, name='toffee')        #     var toffee >= 0;
m.set_objective(0.25*choco + 0.75*toffee,           #     maximize profit =
                sense=so.MAX,                       #        0.25*choco + 0.75*toffee;
                name='profit')                      #
m.add_constraint(15*choco + 40*toffee <= 27000,     #     con process1:
                 name='process1')                   #        15*choco + 40*toffee <= 27000;
m.add_constraint(56.25*toffee <= 27000,             #     con process2:
                 name='process2')                   #        56.25*toffee <= 27000;
m.add_constraint(18.75*choco <= 27000,              #     con process3:
                 name='process3')                   #        18.75*choco <= 27000;
m.add_constraint(12*choco + 50*toffee <= 27000,     #     con process4:
                 name='process4')                   #        12*choco + 50*toffee <= 27000;
result = m.solve()                                  #     solve;
print(so.get_solution_table(choco, toffee))         #     print choco toffee;
                                                    #  quit;
```

As you can see, nearly every line of this Python code has a corresponding line of PROC OPTMODEL code.

SYNTAX

The syntax of the *sasoptpy* package should be natural for Python programmers, and existing Python data structures are easily used in the modeling. You can use the Python list structure in the same way that you use sets in PROC OPTMODEL. Pandas DataFrames and Series objects and Python dictionaries are easy ways to define the problem data. There are specific approaches for adding variables, objectives, and constraints. The options for the solvers are the same as the options for the actions and are directly passed as a dictionary. The *sasoptpy* package can currently use the LP and MILP solvers, and it might soon use the QP solver.

The most common Python methods for writing a model are the **add_variable** and **add_variables** methods to add new variables to the model, the **add_objective** method to add an objective function to minimize or maximize, and the **add_constraint** and **add_constraints** methods to add new constraints to the model.

The syntax for writing expressions for objectives and constraints is familiar to Python programmers, with the exception of summations. Although you can use Python's native **sum** function in *sasoptpy* expressions, it is not efficient for summing over variables or expressions. The *sasoptpy* **quick_sum** function is more efficient for processing summations of expressions. Also, the **VariableGroup.sum** method is a convenient way to sum all variables, or a subset of variables, in a variable group without coefficients.

One difference to note between the *sasoptpy* package and PROC OPTMODEL is that PROC OPTMODEL allows the declaration of the model to be completely independent from the data, whereas *sasoptpy* does not at this time. You

can use *sasoptpy* to construct only concrete models. This means, for example, that the elements of a set must be known before you declare a variable over it, and the values of coefficients in a constraint must be known when the constraint is declared. The following examples demonstrate how you can use data in *sasoptpy*.

You can use data from a Python list:

```
a = [1,2,4,8,16]
S = range(len(a))
X = m.add_variables(S, name='X')
m.set_objective(so.quick_sum(a[i] * X[i] for i in S), sense=so.MIN)
```

You can use data from a Python dictionary:

```
cost = {'Steel': 100, 'Wood': 200, 'Wire': 30}
MATERIALS = cost.keys()
PurchaseAmount = m.add_variables(MATERIALS, name='PurchaseAmount')
m.set_objective(so.quick_sum(cost[i] * PurchaseAmount[i] for i in MATERIALS), sense=so.MIN)
```

You can use data from a Pandas DataFrame:

```
cookingdata = pd.DataFrame([
    ['Process1', 27000, 15, 40],
    ['Process2', 27000, 0, 56.25],
    ['Process3', 27000, 18.75, 0],
    ['Process4', 27000, 12, 50]
    ], columns=['Process', 'Time', 'Chocolate', 'Toffee']).set_index(['Process'])
choco = m.add_variable(lb=0, name='choco')
toffee = m.add_variable(lb=0, name='toffee')
m.add_constraints(cookingdata.at[process, 'Chocolate']*choco
                  + cookingdata.at[process, 'Toffee']*toffee
                  <= cookingdata.at[process, 'Time'] for process in cookingdata.index.values)
```

You can modify the model coefficients at any time, including after you call the solver. You add variables or constraints by using the same methods as before the solve, but the ability to remove variables or constraints is not currently available. You can have multiple models available for manipulation at the same time when you use *sasoptpy*.

HOW THE *sasoptpy* MODELING PACKAGE WORKS

Currently, the only way to access the SAS Optimization solvers from Python is to call the solver action and provide it with an MPS-format data table; this process is described in *SAS Optimization: Mathematical Optimization Programming Guide*. The MPS-format data table is a way to store the problem instance on the CAS server, but it is difficult to create manually. The *sasoptpy* package provides Python functions to users for creating an MPS-format data table, without the user even needing to know that the data table exists. It uses Python's operator overloading feature to parse expressions that are written in a natural form.

The *sasoptpy* package benefits from native Python structures and the Pandas library for fast deployment of models. All the model components are stored inside Python objects until the **`solve`** method is called. When you call the **`solve`** method, *sasoptpy* converts the objects and expressions into a Pandas DataFrame in the MPS-format data table and uploads it to the CAS server. Then it calls the desired solver, passing the newly made data table and any parameters that you include. The parameters are passed as a dictionary—the same way that they would be in the Python interface for the **`solveLp`** or **`solveMilp`** action in the **`optimization`** action set. For example, you can set initial values for the primal and dual variables by using the **`primalIn`** and **`dualIn`** parameters. When a result is returned, you can access primal and dual solution tables, set by the **`primalOut`** and **`dualOut`** parameters, by using the **`get_solution`** method. Data in CAS data tables can be used by *sasoptpy* only if the data are fetched (copied from the CAS server to the client) to Python first.

EXAMPLE: KIDNEY EXCHANGE

The kidney exchange problem was presented along with PROC OPTMODEL code by Galati (2015). That source describes the problem this way:

> Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs A and B are in this situation, but donor A is

compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap in which donor A gives to recipient B and donor B gives to recipient A. More generally, with n such incompatible donor-recipient pairs you can sometimes do an n-way swap.

The objective is to maximize the weighted sum of donor-recipient pairs who are matched for an exchange. This example shows you how to use the *sasoptpy* package to model and solve the problem, and it shows you how other Python packages can use the results to visualize the solution. Several features are shown in this example, but you can refer to the documentation to see all the options and possible parameter values.

IMPORT PYTHON PACKAGES

The first step is to import the necessary Python packages. You need the **CAS** function from the SAS SWAT package as described in *Getting Started with SAS Viya for Python* to connect to the CAS server. You also need the *sasoptpy* package to write the model. This example uses random data, so the random package needs to be imported. The NetworkX and Matplotlib packages are used to plot the data and solution for the example.

```
from swat import CAS
import sasoptpy as so
import random
import networkx as nx
import matplotlib.pyplot as plt
```

CONNECT TO CAS

The next step is to connect to the CAS server:

```
s = CAS(host, port, username, password)
```

CREATE AND VISUALIZE DATA

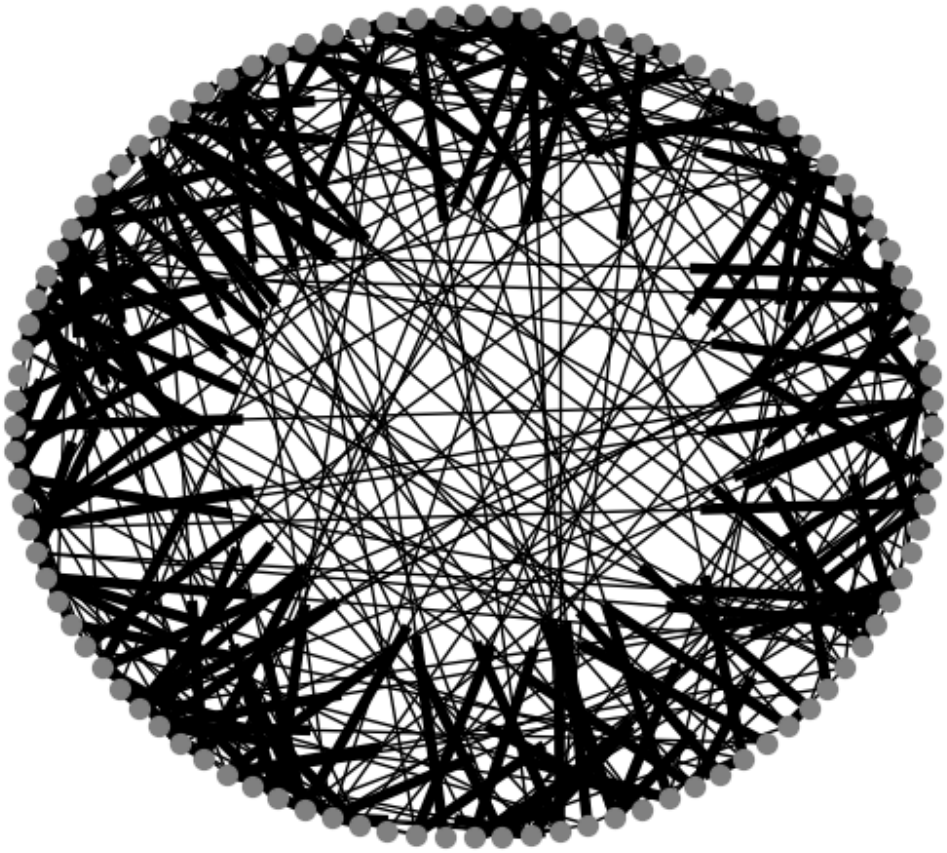
The random data are then created using 100 recipient-donor pairs that have a 2% chance of compatibility with each other pair. The random seed is set for reproducibility, and the **ARCS** dictionary is used to indicate whether a donor is compatible with a recipient in a different pair. The **max_length** parameter is defined to limit cycles to a length of 10, because large cycles are impractical. These statements create the data:

```
n = 100
p = 0.02
max_length = 10
random.seed(1)
ARCS = {}
for i in range(n):
    for j in range(n):
        if i != j:
            if random.random() < p:
                ARCS[i, j] = random.random()
```

Figure 1 shows a plot of the corresponding directed graph. The NetworkX package uses a thicker line at the destination end of the arc to indicate the arc's direction. These statements create and plot the graph:

```
G = nx.DiGraph()
G.add_nodes_from(range(n))
G.add_edges_from(ARCS)
pos = nx.circular_layout(G)
nx.draw_networkx_nodes(G, pos, with_labels=False, node_size=50, node_color='gray')
nx.draw_networkx_edges(G, pos)
plt.axis('off')
plt.show()
```

Figure 1 Full Compatibility Graph



DEFINE THE MODEL

Next, the *sasoptpy* package is used to define the model. First, the model object is initialized with a name and a CAS session:

```
m = so.Model("kidney_exchange", session=s)
```

Declare Sets

Then, in the following code, the Python set object **NODES** is defined according to the keys of the **ARCS** dictionary, which takes the integer values between 0 and 99. The **MATCHINGS** set contains the names of each cycle in the solution. Each cycle contains at least two nodes, so the solution has at most 50 cycles. They are defined as the integers 1 through 50.

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

Declare Variables

Then the **UseNode**, **UseArc**, and **Slack** variables are declared, as the following code shows. The **UseNode** variable indicates whether a particular node is in a particular matching. The **UseArc** variable indicates whether a particular arc is in a particular matching. The **Slack** variable indicates whether a particular node is not in any matching.

Note that the first arguments of the **add_variables** method are the sets that the variable is defined over. Then the optional **vartype** parameter is used to indicate whether the variable is continuous, binary, or integer. The optional **name** parameter is used as a prefix for variables in the same group. You can also use the specified name to access the object by using the **get_obj_by_name** function.

```
UseNode = m.add_variables(NODES, MATCHINGS, vartype=so.BIN, name="UseNode")
UseArc = m.add_variables(ARCS, MATCHINGS, vartype=so.BIN, name="UseArc")
Slack = m.add_variables(NODES, vartype=so.BIN, name="Slack")
```

Define Objective

Setting the objective shows how you can write expressions. The first argument of the **set_objective** method is the expression, the **name** parameter specifies a name, and the **sense** parameter defines the objective as minimization or maximization. This objective maximizes the total weight of the used arcs:

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

Define Constraints

You can add constraints in groups over sets by using the **add_constraints** method. The first argument is the equality or inequality, which can include a **for** statement if it is over a set. The second argument is the name for the group of constraints.

The **Node_Packing** constraint requires each node to appear in no more than one matching. The **Donate** constraint requires there to be no more than one recipient for each donor. The **Receive** constraint requires there to be no more than one donor for each recipient. The **Cardinality** constraint prevents long matching cycles.

The **VariableGroup.sum** method for variable groups is used for efficient summations over index sets that have **'*'** in the index's position. Using this method allows the expression to be written more compactly than using the **quick_sum** function, which must be used in the objective because of the coefficients. The alternative is also shown for the **Node_Packing** constraint in a comment in these statements:

```
Node_Packing = m.add_constraints(
    (UseNode.sum(i, '*') + Slack[i] == 1 for i in NODES),
    name="node_packing")
# Alternative summation method
# Node_Packing = m.add_constraints(
#   (so.quick_sum(UseNode[i, k] for k in MATCHINGS) + Slack[i] == 1 for i in NODES),
#   name="node_packing")
Donate = m.add_constraints(
    (UseArc.sum(i, '*', k) == UseNode[i, k] for i in NODES for k in MATCHINGS),
    name="donate")
Receive = m.add_constraints(
    (UseArc.sum('*', j, k) == UseNode[j, k] for j in NODES for k in MATCHINGS),
    name="receive")
Cardinality = m.add_constraints(
    (UseArc.sum('*', '*', k) <= max_length for k in MATCHINGS),
    name="cardinality")
```

Solve with DECOMP

If the **Node_Packing** constraint is relaxed, the problem becomes decomposable into independent subproblems, so the model is a good candidate for the DECOMP algorithm in the MILP solver. The model is separable by matching, so as the following code shows, the block assignment for each constraint is set to k. The solver is then called using DECOMP, with the presolver level set to BASIC for the model to maintain symmetry.

```
for i in NODES:
    for k in MATCHINGS:
        Donate[i, k].set_block(k)

for j in NODES:
    for k in MATCHINGS:
        Receive[j, k].set_block(k)
```

```
for k in MATCHINGS:
    Cardinality[k].set_block(k)

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

The output from the solver action is displayed after the **solve** method is called. However, the following *sasoptpy* package notes precede the solver output. The first note indicates that the model has been initialized. Because the block assignments are set for constraints, *sasoptpy* creates a data table, called "BLOCKSTABLE," to pass this information to the solver. The *sasoptpy* output then indicates that it is converting the model to a Pandas DataFrame. It uploads the Pandas DataFrame to the CAS server as a data table with an arbitrary name and adds the **optimization** action set.

```
NOTE: Initialized model kidney_exchange
NOTE: Cloud Analytic Services made the uploaded file available as table BLOCKSTABLE in
caslib CASUSERHDFS(casuser).
NOTE: The table BLOCKSTABLE has been created in caslib CASUSERHDFS(casuser) from binary
data uploaded to Cloud Analytic Services.
NOTE: Converting model kidney_exchange to DataFrame
NOTE: Uploading the problem DataFrame to the server.
NOTE: Cloud Analytic Services made the uploaded file available as table TMPK04UM1ZU in
caslib CASUSERHDFS(casuser).
NOTE: The table TMPK04UM1ZU has been created in caslib CASUSERHDFS(casuser) from binary
data uploaded to Cloud Analytic Services.
NOTE: Added action set 'optimization'.
```

The next part of the log is created by the solver and is not shown here. The following notes, which are displayed after the objective value, indicate that *sasoptpy* drops (removes from the CAS session) the tables that it creates:

```
NOTE: Objective = 28.060783917.
NOTE: Cloud Analytic Services dropped table TMPK04UM1ZU from caslib CASUSERHDFS(casuser).
NOTE: Cloud Analytic Services dropped table BLOCKSTABLE from caslib CASUSERHDFS(casuser).
```

VIEW SOLUTION

When you use the **get_solution_table** function, as in the following code, you can get the values of the **UseArc** variable in a Pandas DataFrame.The rest of the code after the solution is extracted does not use *sasoptpy*—only the packages that you need in order to generate plots. Then a list of the used arcs and a list of the lists of nodes for each cycle are created; they are used for a new NetworkX graph that displays only the used arcs.

```
UseArcDataFrame = so.get_solution_table(UseArc)

usedarcs = []
cycleassign = {}
cyclenodes = []
for index, row in UseArcDataFrame.iterrows():
    if row['UseArc'] > 0.5:
        node_from = index[0]
        node_to = index[1]
        matching = index[2]
        usedarcs.append((node_from, node_to))
        if matching in cycleassign:
            cyclenodes[cycleassign[matching]].append(node_from)
        else:
            cycleassign[matching] = len(cyclenodes)
            cyclenodes.append([node_from])

print(UseArcDataFrame)
print(usedarcs)
```

The beginning of the printed Pandas DataFrame output shows the values of the **UseArc** variable, starting with the first edge from 0 to 14 for each element of **MATCHINGS**. The value of **UseArc** is 1 when the **MATCHINGS** element is 1, so that edge is in the first cycle and appears in the **usedarcs** list output.

```
           UseArc
1  2  3
0  14 1      1.0
0  14 2      0.0
```

```
0  14 3      0.0
0  14 4      0.0
0  14 5      0.0
0  14 6      0.0
0  14 7      0.0
0  14 8      0.0
0  14 9      0.0
0  14 10     0.0
0  14 11     0.0
...          ...
99 76 43     0.0
99 76 44     0.0
99 76 45     0.0
99 76 46     0.0
99 76 47     0.0
99 76 48     0.0
99 76 49     0.0

[(0, 14), (1, 77), (2, 0), (5, 62), (8, 63), (10, 37), (11, 29), (12, 26), (13, 59),
(14, 42), (17, 49), (21, 85), (25, 13), (26, 47), (27, 8), (29, 5), (31, 33), (32, 10
), (33, 43), (34, 92), (36, 55), (37, 76), (38, 88), (40, 38), (42, 21), (43, 87), (4
4, 17), (46, 32), (47, 44), (49, 56), (51, 46), (55, 93), (56, 70), (59, 82), (62, 1)
, (63, 71), (64, 51), (66, 69), (69, 64), (70, 12), (71, 11), (75, 98), (76, 66), (77
, 27), (82, 94), (85, 40), (87, 34), (88, 2), (92, 36), (93, 31), (94, 75), (98, 25)]
```

The following statements assign colors so that the nodes and arcs in each cycle are the same color and unused nodes are gray:

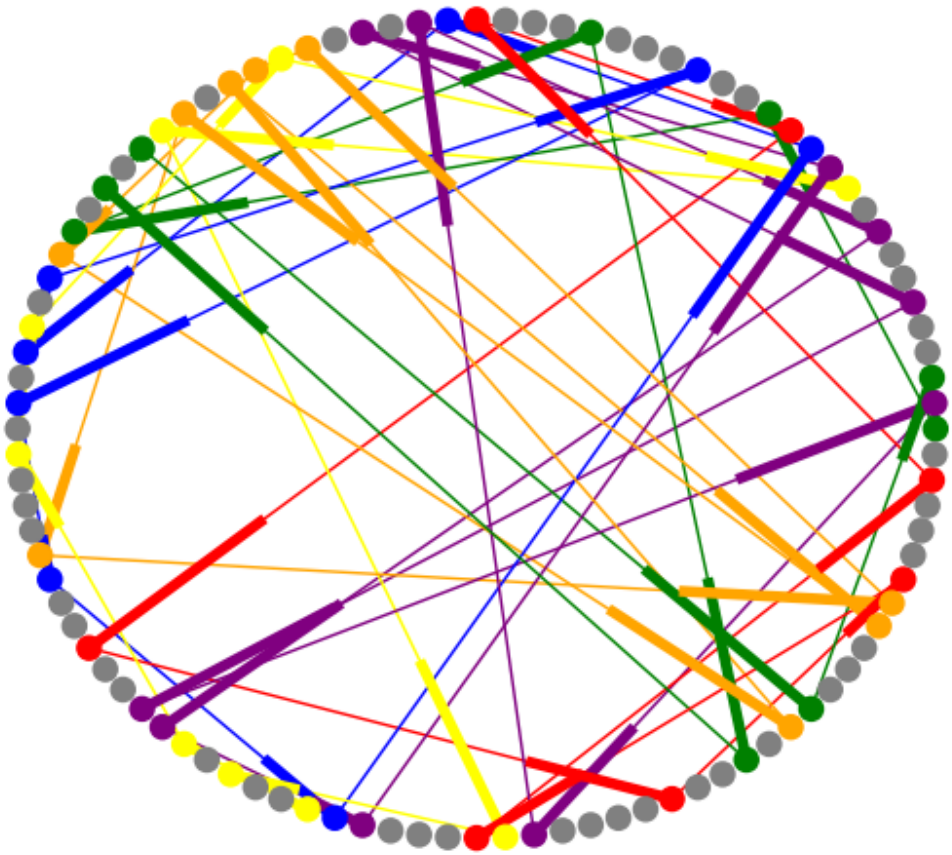
```
cyclenodes.sort(key=len)
nodecolors = ['gray'] * n
colorlist = ['red', 'blue', 'green', 'yellow', 'orange', 'purple']
coloridx = 0
for cycle in cyclenodes:
    color = colorlist[coloridx%len(colorlist)]
    for i in cycle:
        nodecolors[i] = color
    coloridx += 1

edgecolors = [0] * len(usedarcs)
edgenumber = 0
for i,j in usedarcs:
    edgecolors[edgenumber] = nodecolors[i]
    edgenumber += 1
```

The following statements plot the solution by using NetworkX, as shown in Figure 2:

```
G = nx.DiGraph()
G.add_nodes_from(range(n))
G.add_edges_from(usedarcs)
nx.draw_networkx(G, pos, edges=G.edges, nodes=G.nodes, node_color=nodecolors,
                 edge_color=edgecolors, with_labels=False, node_size=75)
plt.axis('off')
plt.show()
```

Figure 2 Solution Graph

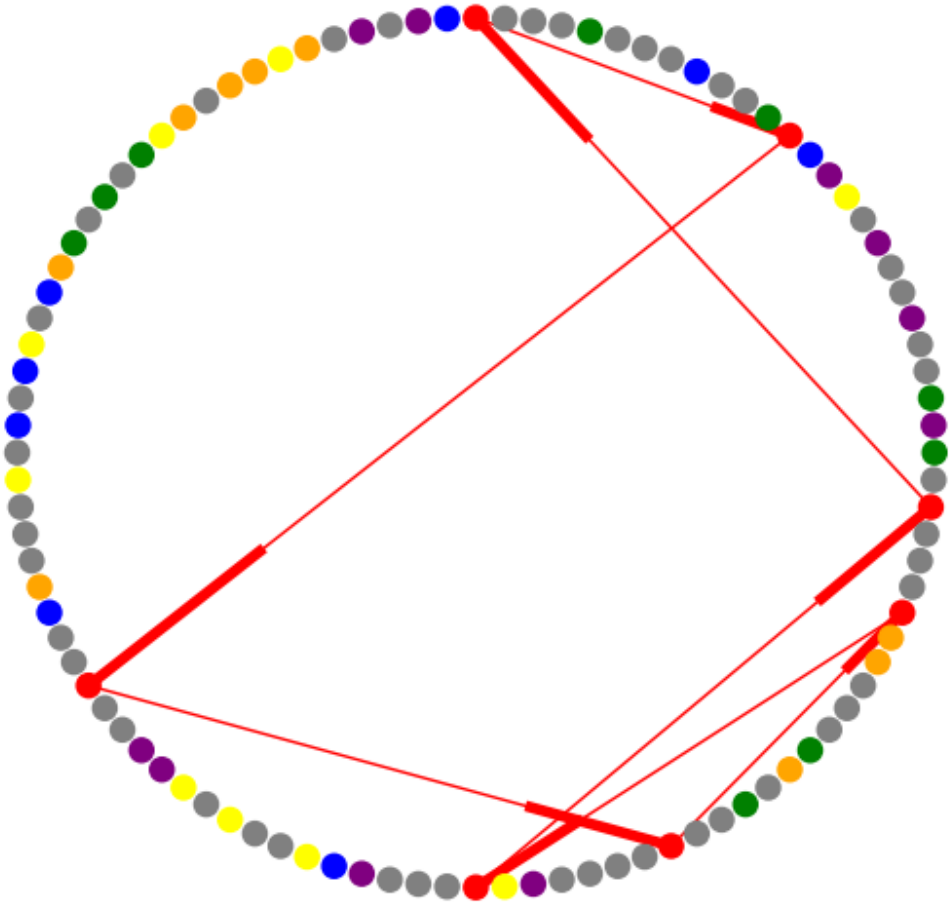


The following statements plot an additional graph that shows an individual cycle, as shown in Figure 3:

```
singlecyclearcs = [arcs for arcs in usedarcs if arcs[0] in cyclenodes[0]]
edgecolor = nodecolors[cyclenodes[0][0]]

G=nx.DiGraph()
G.add_nodes_from(range(n))
G.add_edges_from(singlecyclearcs)
nx.draw_networkx(G, pos, edges=G.edges, nodes=G.nodes, node_color=nodecolors,
                 edge_color=edgecolor, with_labels=False, node_size=75)
plt.axis('off')
plt.show()
```

Figure 3 Single-Cycle Graph



SUMMARY

This paper shows a new way to use SAS Optimization solvers from a Python interface. It combines the advantages of Python as a popular programming environment for data scientists and the convenience of modeling your optimization problem in an intuitive way. The ability to use the *sasoptpy* modeling package with familiar Python packages, as demonstrated using Matplotlib and NetworkX, is also an attractive feature for Python programmers. The open-source package is available at https://github.com/sassoftware/sasoptpy. For further reading about using Python with SAS Viya, see Smith and Meng (2017).

REFERENCES

Galati, M. (2015). "The Kidney Exchange Problem." February. https://blogs.sas.com/content/operations/2015/02/06/the-kidney-exchange-problem/.

SAS Institute Inc. (2017a). *Getting Started with SAS Viya 3.3 for Python*. Cary, NC: SAS Institute Inc. http://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.3&docsetId=caspg3&docsetTarget=titlepage.htm&locale=en.

SAS Institute Inc. (2017b). *An Introduction to SAS Viya 3.3 Programming*. Cary, NC: SAS Institute Inc. http://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.3&docsetId=pgmdiff&docsetTarget=n1t409khqsu0n8n103122kk0bfzn.htm&locale=en.

SAS Institute Inc. (2017c). *SAS Optimization 8.2: Mathematical Optimization Procedures*. Cary, NC: SAS Institute Inc. http://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.3&docsetId=casmopt&docsetTarget=titlepage.htm&locale=en.

SAS Institute Inc. (2017d). *SAS Optimization 8.2: Mathematical Optimization Programming Guide*. Cary, NC: SAS Institute Inc. http://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.3&docsetId=casactmopt&docsetTarget=titlepage.htm&locale=en.

SAS Institute Inc. (2017e). *SAS/OR 14.3 User's Guide: Mathematical Programming*. Cary, NC: SAS Institute Inc. http://go.documentation.sas.com/?docsetId=ormpug&docsetTarget=titlepage.htm&docsetVersion=14.3&locale=en.

Smith, K. D., and Meng, X. (2017). *SAS Viya: The Python Perspective*. Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors:

Jared Erickson
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
jared.erickson@sas.com

Sertalp B. Cay
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
sertalp.cay@sas.com