pyomo_simplemodel Documentation

Release 1.0.1

pyomo_simplemodel

Jul 23, 2020

Contents:

The **pyomo_simplemodel** package is software for modeling and solving optimization problems. This package is derived from Pyomo, and it defines the class **SimpleModel** that illustrates how Pyomo can be used in a simple, less object-oriented manner. Specifically, this class mimics the modeling style supported by PuLP:

Feature	PuLP	SimpleModel
LP/MILP	YES	YES
NLP/MINLP	NO	YES
Column-wise	YES	NO

SimpleModel vs Pyomo

SimpleModel is not meant to serve as a replacement for Pyomo. While SimpleModel only represents problems with a simple, unstructured representation, Pyomo's modeling components support structured, hierarchical models that are suitable for complex applications.

The following sections illustrate similarities and differences between SimpleModel, PuLP and regular Pyomo models. First, the knapsack problem is used to illustrate that these packages can be used in a similar manner on simple applications. Next, the soda can problem illustrates that SimpleModel can represent nonlinear problems that cannot be modeled with PuLP. Finally, the newvendor problem is used to illustrate three different modeling representations: unstructured models, structured models and hierarchical models. SimpleModel and PuLP have unstructured models, while Pyomo supports all three modeling representations.

CHAPTER 1

Comparing SimpleModel, PuLP and Pyomo

This section illustrates differences between SimpleModel, PuLP and regular Pyomo models on the **knapsack problem**. This problem can be represented as a integer program, which all three of these modeling tools can easily represent.

1.1 Knapsack Problem

The Knapsack Problem considers the problem of selecting a set of items whose weight is not greater than a specified limit while maximizing the total value of the selected items. This problem is inspired by the challenge of filling a knapsack (or rucksack) with the most valuable items that can be carried.

A common version of this problem is the **0-1 knapsack problem**, where each item is distinct and can be selected once. Suppose there are n items with positive values $v_1, \ldots, v_n$ and weights $w_1, \ldots, w_n$. Let $x_1, \ldots, x_n$ be decision variables that can take values 0 or 1. Let *W* be the weight capacity of the knapsack.

The following optimization formulation represents this problem as an integer program:

$$\begin{array}{ll} \max & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} & \sum_{i=1}^{n} w_i x_i \leq W \\ & x_i \in \{0, 1\} \end{array}$$

The following sections illustrate how this optimization problem can be formulated with (1) SimpleModel, (2) PuLP, and (3) Pyomo.

1.2 SimpleModel Formulation

The following script executes the following steps to create and solve a knapsack problem:

1. Import `pyomo_simplemodel`
2. Construct a `SimpleModel` class
3. Declare variables, the objective and the constraint
4. Perform optimization

5. Summarize the optimal solution

```
# knapsack.py

from pyomo_simplemodel import *

v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
limit = 14
items = list(sorted(v.keys()))

# Create model
m = SimpleModel(maximize=True)

# Variables
x = m.var('m', items, within=Binary)

# Objective
m += sum(v[i]*x[i] for i in items)

# Constraint
m += sum(w[i]*x[i] for i in items) <= limit


# Optimize
status = m.solve('glpk')

# Print the status of the solved LP
print("Status = %s" % status.solver.termination_condition)

# Print the value of the variables at the optimum
for i in items:
    print("%s = %f" % (x[i], value(x[i])))

# Print the value of the objective
print("Objective = %f" % value(m.objective()))

```

In this example, the model object `m` is used to manage the problem definition. Decision variables are declared with the `var()` method, the objective and constraint are added with the += operator, and the `solve()` method is used to perform optimization. After optimization, the solution is stored in the variable objects, and the objective value can be accessed with using the `objective()` method.

1.3 PuLP Formulation

The following script executes the same steps as above to create and solve a knapsack problem using PuLP:

```
# knapsack-pulp.py

from pulp import *

v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
limit = 14
items = list(sorted(v.keys()))

```

(continues on next page)

(continued from previous page)

```
# Create model
m = LpProblem("Knapsack", LpMaximize)

# Variables
x = LpVariable.dicts('x', items, lowBound=0, upBound=1, cat=LpInteger)

# Objective
m += sum(v[i]*x[i] for i in items)

# Constraint
m += sum(w[i]*x[i] for i in items) <= limit

# Optimize
m.solve()

# Print the status of the solved LP
print("Status = %s" % LpStatus[m.status])

# Print the value of the variables at the optimum
for i in items:
    print("%s = %f" % (x[i].name, x[i].varValue))

# Print the value of the objective
print("Objective = %f" % value(m.objective))
```

This script is *very* similar to the SimpleModel script. Both scripts declare a problem class that is used to declare variables, the objective and constraint, and to perform optimization.

1.4 Pyomo Formulation

The following script executes the same steps as above to create and solve a knapsack problem using Pyomo:

```
# knapsack-pyomo.py

from pyomo.environ import *

v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
limit = 14
items = list(sorted(v.keys()))

# Create model
m = ConcreteModel()

# Variables
m.x = Var(items, within=Binary)

# Objective
m.value = Objective(expr=sum(v[i]*m.x[i] for i in items), sense=maximize)

# Constraint
m.weight = Constraint(expr=sum(w[i]*m.x[i] for i in items) <= limit)
```

(continues on next page)

(continued from previous page)

```
# Optimize
solver = SolverFactory('glpk')
status = solver.solve(m)

# Print the status of the solved LP
print("Status = %s" % status.solver.termination_condition)

# Print the value of the variables at the optimum
for i in items:
    print("%s = %f" % (m.x[i], value(m.x[i])))

# Print the value of the objective
print("Objective = %f" % value(m.value))
```

This script is similar to the SimpleModel and PuLP scripts, but Pyomo models are created with an object-oriented design. Thus, elements of the optimization problem are declared with variable, objective and constraint components, which are Pyomo objects. As a consequence, the objective and constraint expressions reference variable components within the model (e.g. `m.x`) instead of variable objects directly (e.g. `x`). Thus, modeling in Pyomo is more verbose (especially when long model names are used).

CHAPTER 2

Modeling Nonlinear Problems

This section illustrates differences between SimpleModel and regular Pyomo models on a simple nonlinear problem. PuLP is omitted from this comparison because it cannot represent nonlinear problems.

2.1 Soda Can Problem

Finding the optimal dimensions of a soda can is a simple nonlinear optimization problem. We consider an idealized soda can that is represented as a cylinder with radius *r* and height *h*. The problem is to find the radius and height that minimizes the surface area of the cylinder while keeping a fixed volume. Here, the surface area of the cylinder approximates the amount of aluminum needed for a soda can, so this problem can be used to predict the miminum amount of aluminum needed to hold a given volume.

The surface area of a cylinder is

$$2\pi r(r+h)$$

A standard soda can is 12 oz or 355 ml. Thus, we have the constraint

$$\pi r^2 h = 355$$

Thus, we have the following optimization representation for this problem:

$$\begin{array}{ll} \min & 2\pi r(r+h) \\ \text{s.t.} & \pi r^2 h = 355 \\ & r \geq 0 \\ & h \geq 0 \end{array}$$

This is a nonlinear problem, so it cannot be formulated with PuLP. The following sections illustrate how this optimization problem can be formulated and solved with SimpleModel and Pyomo.

2.2 SimpleModel Formulation

The following script executes the following steps to create and solve the soda can problem:

1. Import `pyomo_simplemodel`
2. Construct a `SimpleModel` class
3. Declare variables, the objective and the constraint
4. Perform optimization
5. Summarize the optimal solution

```
# sodacan.py

from pyomo_simplemodel import *
from math import pi

m = SimpleModel()

r = m.var('r', bounds=(0,None))
h = m.var('h', bounds=(0,None))

m += 2*pi*r*(r + h)
m += pi*h*r**2 == 355

status = m.solve("ipopt")

print("Status = %s" % status.solver.termination_condition)

print("%s = %f" % (r, value(r)))
print("%s = %f" % (h, value(h)))
print("Objective = %f" % value(m.objective()))

```

In this example, the model object `m` is used to manage the problem definition. Decision variables are declared with the `var()` method, the objective and constraint are added with the += operator, and the `solve()` method is used to perform optimization. After optimization, the solution is stored in the variable objects, and the objective value can be accessed with using the `objective()` method.

2.3 Pyomo Formulation

The following script executes the same steps as above to create and solve the soda can problem using Pyomo:

```
# sodacan-pyomo.py

from pyomo.environ import *
from math import pi

m = ConcreteModel()

m.r = Var(bounds=(0,None))
m.h = Var(bounds=(0,None))

m.o = Objective(expr=2*pi*m.r*(m.r + m.h))
m.c = Constraint(expr=pi*m.h*m.r**2 == 355)

solver = SolverFactory('ipopt')
status = solver.solve(m)

```

(continues on next page)

(continued from previous page)

```
print("Status = %s" % status.solver.termination_condition)

print("%s = %f" % (m.r, value(m.r)))
print("%s = %f" % (m.h, value(m.h)))
print("Objective = %f" % value(m.o))
```

This script is similar to the SimpleModel script, but Pyomo models are created with an object-oriented design. Thus, elements of the optimization problem are declared with variable, objective and constraint components, which are Pyomo objects. As a consequence, the objective and constraint expressions reference variable components within the model (e.g. `m.x`) instead of variable objects directly (e.g. `x`).

CHAPTER 3

Unstructured, Structured and Block Formulations

This section illustrates differences between SimpleModel, PuLP and regular Pyomo models on a problem with more complex structure. The **newsvendor problem** is used to illustrate three different modeling representations that are supported by these modeling tools:

unstructured The model stores a *list* of objectives and constraints expressions.

structured The model stores named objectives and constraints. Each of these named components *map* index values to expressions.

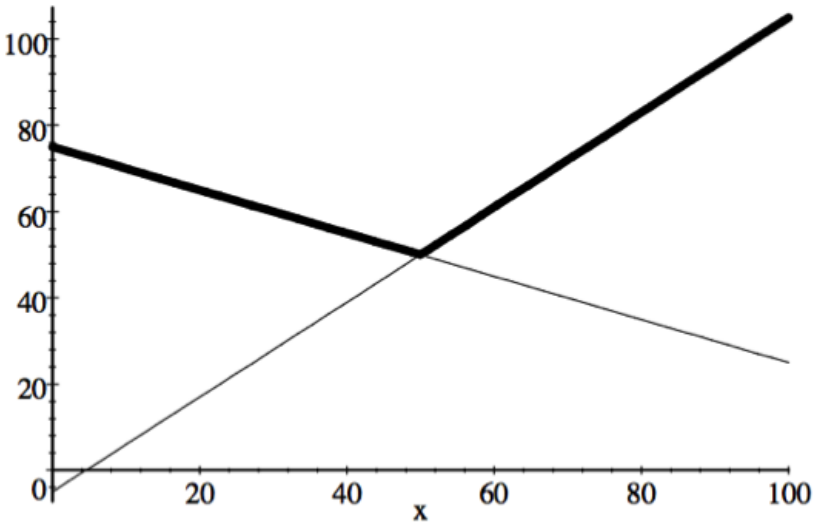
hierarchical The model stores named *block* components, each of which stores named components, including variables, objectives and constraints.

3.1 Newsvendor Problem

The Newsvendor Problem considers the problem of determining optimal inventory levels. Given fixed prices and an uncertain demand, the problem is to determine inventory levels that maximize the expected profit for the newsvendor.

The following formulation is adapted from Shapiro and Philpott [ShaPhi]. A company has decided to order a quantity x of a product to satisfy demand d. The per-unit cost of ordering is c, and if demand d is greater than x, $d>x$, then the back-order penalty is b per unit. If demand is less than production, $d<x$, then a holding cost h is incurred for unused product.

The objective is to minimize the *total cost*: $\max\{(c-b)x+bd,(c+h)x-hd\}$. For example, suppose we have $c=1$, $b=1.5$, $h=0.1$, and $d=50$. Then the following figure illustrates the total cost:



In general, the ordering decision is made before a realization of the demand is known. The deterministic formulation corresponds to a single scenario taken with probability one:

$$\begin{array}{ll} \min_{x,y} & y \\ \text{s.t.} & y \geq (c-b)x + bd \\ & y \geq (c+h)x - hd \\ & x \geq 0 \end{array}$$

Now suppose we model the distribution of possible demands with scenarios $d_1, \ldots, d_K$ each with equal probability ($p_k = 0.2$). Then the following formulation minimizes the expected value of the total cost over these scenarios:

$$\begin{array}{lllll} \min_{x,y_1,\ldots,y_K} & \sum_k p_k y_k & & & \\ \text{s.t.} & y_k \geq (c-b)x + bd_k & k = 1,\ldots,K & \textit{(Demand is greater)} \\ & y_k \geq (c+h)x - hd_k & k = 1,\ldots,K & \textit{(Demand is less)} \\ & x \geq 0 \end{array}$$

This is a linear problem, so it can be formulated with SimpleModel, PuLP and Pyomo.

Since the two constraints are indexed from $1, \ldots, K$, we can group them together into a single block, which itself is indexed from $1, \ldots, K$.

$$\begin{array}{ll} \min_{x,y_1,\ldots,y_K} & \sum_k p_k y_k \\ \text{s.t.} & \left\{ \begin{array}{l} y_k \geq (c-b)x + bd_k \\ y_k \geq (c+h)x - hd_k \end{array} \right\} \quad k = 1,\ldots,K \\ & x \geq 0 \end{array}$$

Below, we show formulations for SimpleModel, PuLP, and Pyomo. The SimpleModel and PuLP models illustrate *unstructured* representations. The first Pyomo formulation illustrates an *unstructured* representation, where constraints are stored in a list. The second Pyomo formulation illustrates a *structured* representation, which corresponds to the first formulation above. The final Pyomo formulation illustrates a *hierarchical* representation, using the **Block** component to structure the model representation in a modular manner.

3.2 SimpleModel Formulation

The following script creates and solves a linear program for the newsvendor problem using SimpleModel:

```
# newsvendor.py

from pyomo_simplemodel import *

c=1.0
b=1.5
h=0.1
d = {1:15, 2:60, 3:72, 4:78, 5:82}

scenarios = range(1,6)

m = SimpleModel()
x = m.var('x', within=NonNegativeReals)
y = m.var('y', scenarios)

for i in scenarios:
  m += y[i] >= (c-b)*x + b*d[i]
  m += y[i] >= (c+h)*x - h*d[i]

m += sum(y[i] for i in scenarios)/5.0

status = m.solve("glpk")

print("Status = %s" % status.solver.termination_condition)

print("%s = %f" % (x, value(x)))
for i in y:
    print("%s = %f" % (y[i], value(y[i])))
print("Objective = %f" % value(m.objective()))

```

There are two key things to note about this model. First, the model simply consists of a list of constraints. Second, the `y` variable is indexed to represent the total cost for the different scenarios.

3.3 PuLP Formulation

The following script creates and solves a linear program for the newsvendor problem using PuLP:

```
# newsvendor-pulp.py

from pulp import *

c=1.0
b=1.5
h=0.1
d = {1:15, 2:60, 3:72, 4:78, 5:82}

scenarios = range(1,6)

M = LpProblem("Newsvendor")

x = LpVariable('x', lowBound=0)
y = LpVariable.dicts('y', scenarios)

for i in scenarios:
```

(continues on next page)

(continued from previous page)

```
  M += y[i] >= (c-b)*x + b*d[i]
  M += y[i] >= (c+h)*x - h*d[i]

M += sum(y[i] for i in scenarios)/5.0

M.solve()

print("Status = %s" % LpStatus[M.status])

print("%s = %f" % (x.name, value(x.varValue)))
for i in scenarios:
    print("%s = %f" % (y[i].name, y[i].varValue))
print("Objective = %f" % value(M.objective))

```

As with the SimpleModel formulation, the model consists of a list of constraints, and the `y` variable is indexed.

3.4 Pyomo Formulations

The following script creates and solves a linear program for the newsvendor problem using Pyomo:

```
# newsvendor-pyomo1.py

from pyomo.environ import *

c=1.0
b=1.5
h=0.1
d = {1:15, 2:60, 3:72, 4:78, 5:82}

scenarios = range(1,6)

M = ConcreteModel()
M.x = Var(within=NonNegativeReals)
M.y = Var(scenarios)

M.c = ConstraintList()
for i in scenarios:
  M.c.add( M.y[i] >= (c-b)*M.x + b*d[i] )
  M.c.add( M.y[i] >= (c+h)*M.x - h*d[i] )

M.o = Objective(expr=sum(M.y[i] for i in scenarios)/5.0)

solver = SolverFactory('glpk')
status = solver.solve(M)

print("Status = %s" % status.solver.termination_condition)

print("%s = %f" % (M.x, value(M.x)))
for i in scenarios:
    print("%s = %f" % (M.y[i], value(M.y[i])))
print("Objective = %f" % value(M.o))

```

This model uses the `ConstraintList` component to store a list of constraints, and the `y` variable is indexed. Thus,

this model provides an *unstructured* representation that is similar to models generated with SimpleModel and PuLP.

The following script uses Pyomo to create and solve the newsvendor problem, using a *structured* representation:

```
# newsvendor-pyomo2.py

from pyomo.environ import *

c=1.0
b=1.5
h=0.1
d = {1:15, 2:60, 3:72, 4:78, 5:82}

scenarios = range(1,6)

M = ConcreteModel()
M.x = Var(within=NonNegativeReals)
M.y = Var(scenarios)

def greater_rule(M, i):
  return M.y[i] >= (c-b)*M.x + b*d[i]
M.greater = Constraint(scenarios, rule=greater_rule)

def less_rule(M, i):
  return M.y[i] >= (c+h)*M.x - h*d[i]
M.less = Constraint(scenarios, rule=less_rule)

def o_rule(M):
    return sum(M.y[i] for i in scenarios)/5.0
M.o = Objective(rule=o_rule)

solver = SolverFactory('glpk')
status = solver.solve(M)

print("Status = %s" % status.solver.termination_condition)

print("%s = %f" % (M.x, value(M.x)))
for i in scenarios:
    print("%s = %f" % (M.y[i], value(M.y[i])))
print("Objective = %f" % value(M.o))
```

The named constraint components `greater` and `less` define two groups of constraints for the model, each of which has the same mathematical form. These named components provide a structured representation for these constraints.

Finally, the following script uses Pyomo to create and solve this problem, using a *hierarchical* representation:

```
# newsvendor-pyomo3.py

from pyomo.environ import *

c=1.0
b=1.5
h=0.1
d = {1:15, 2:60, 3:72, 4:78, 5:82}

scenarios = range(1,6)

M = ConcreteModel()
```

(continues on next page)

(continued from previous page)

```
M.x = Var(within=NonNegativeReals)

def b_rule(B, i):
  B.y = Var()
  B.greater = Constraint(expr=B.y >= (c-b)*M.x + b*d[i])
  B.less    = Constraint(expr=B.y >= (c+h)*M.x - h*d[i])
  return B
M.b = Block(scenarios, rule=b_rule)

def o_rule(M):
    return sum(M.b[i].y for i in scenarios)/5.0
M.o = Objective(rule=o_rule)

solver = SolverFactory('glpk')
status = solver.solve(M)

print("Status = %s" % status.solver.termination_condition)

print("%s = %f" % (M.x, value(M.x)))
for i in scenarios:
    print("%s = %f" % (M.b[i].y, value(M.b[i].y)))
print("Objective = %f" % value(M.o))
```

A block is added for each index $k = 1, \ldots, K$. Each block contains a variable `y`, and the corresponding constraints that define the value of `y`.

Note that the block component `b` is indexed in this formulation while the `y` variable is indexed in the other formulations above. Block components allow Pyomo to support a modular modeling framework where data, variables and constraints can be represented together for each index value. There are several advantages of this approach:

- This model structure is explicit, and it can be exploited by decomposition-based optimization solvers (e.g. the progressive hedging solver in Pyomo).
- Extending and refining models is simpler with blocks. For example, if a multi-dimensional index was needed for this problem, then only the block `b` would need to be modified to reflect that.

CHAPTER 4

Installation

This package can be installed from the GitHub repository using `pip` as follows:

```
pip install git+https://github.com/Pyomo/pyomo_simplemodel
```

Once installed this package can be imported as follows:

```
import pyomo_simplemodel
```

CHAPTER 5

Source Documentation

5.1 SimpleModel

class pyomo_simplemodel.**SimpleModel**(*maximize=False*)

This class illustrates how Pyomo can be used in a simple, less object-oriented manner. Specifically, this class mimics the modeling style supported by PuLP.

This class contains a Pyomo model, and it includes methods that support a simple API for declaring variables, adding objectives and constraints, and solving the model. Optimization results are stored in the variable objects, which are returned to the user.

For example, the following model minimizes the surface area of a soda can while constraining its volume:

```
from pyomo_simplmodel import *
from math import pi

m = SimpleModel()

r = m.var('r', bounds=(0,None))
h = m.var('h', bounds=(0,None))

m += 2*pi*r*(r + h)
m += pi*h*r**2 == 355
```

This model can be solved with the IPOPT solver:

```
status = m.solve("ipopt")
print("Status = %s" % status.solver.termination_condition)
```

The optimum value and decision variables can be easily accesed:

```
print("%s = %f" % (r, value(r)))
print("%s = %f" % (h, value(h)))
print("Objective = %f" % value(m.objective()))
```

Notes

This class illustrates the basic steps in formulating and solving an optimization problem, but it is not meant to serve as a replacement for Pyomo. Pyomo models supports a much richer set of modeling components than simple objectives and constraints. In particular, Pyomo's Block component supports the expression of hierarchical models with nested structure. This class only supports a simple, flat optimization problems.

constraint(*i=1*)
Return the i-th constraint

Parameters **i** (*int*) – The constraint index, which defaults to 1.

Returns An object that defines a constraint

Return type Pyomo constraint object

constraints()
A generator that iterates through all constraints in the model.

Yields *Pyomo constraint object* – An object that defines a constraint

display()
Display the values in the model

objective(*i=1*)
Return the i-th objective

Parameters **i** (*int*) – The objective index, which defaults to 1.

Returns An object that defines an objective

Return type Pyomo objective object

pprint()
Print the equations in the model

solve(*name*, **args*, ***kwargs*)
Optimize the model using the named solver.

Parameters

- **name** (*str*) – The solver name
- ***args** – A variable list of arguments.
- ****kwargs** – A variable list of keyword arguments.

Notes

The arguments and keyword arguments are the same as supported by Pyomo solver objects.

suffix(*name*)
Declare a suffix with the specified name. Suffixes are values returned by the solver, which are typically associated constraints.

Parameters **suffix** (*str*) – The suffix that is returned from the solver.

var(**args*, ***kwds*)
Declare a variable.

Parameters

- ***args** – The first argument is a string for the variable name used by Pyomo. The remaining arguments are assumed to be index sets for the variable.

- ****kwargs** – The keyword arguments are the same as the keyword arguments supported by the Pyomo Var component.

Returns If the variable is not indexed, then the return type is a single Pyomo variable object. If the variable is indexed, then the return type is a dictionary of Pyomo variable objects.

Return type Variable object

5.2 Declaring Variables

By default, model variables are assumed to be unbounded real values. In practice, it is often necessary to specify a more limited set of values. For example, suppose a variable `x` assumes integer values in the range 1 to 7. Then the following declaration would be used:

```
x = m.var('x', bounds=(1,7), within=Integers)
```

The `bounds` keyword specifies the lower and upper bounds for the variable. The `within` keyword indicates the feasible domain for the variable: the set of feasible values that the variable may assume. A variety of objects are defined by Pyomo to specify feasible domains, including:

Binary The set of boolean values

Boolean The set of boolean values

Integers The set of integer values

NegativeIntegers The set of negative integer values

NegativeReals The set of negative real values

NonNegativeIntegers The set of non-negative integer values

NonNegativeReals The set of non-negative real values

NonPositiveIntegers The set of non-positive integer values

NonPositiveReals The set of non-positive real values

PercentFraction The set of real values in the interval [0,1]

PositiveIntegers The set of positive integer values

PositiveReals The set of positive real values

Reals The set of real values

UnitInterval The set of real values in the interval [0,1]

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

6.1 Acknowledgements

This software was supported in part by Sandia National Laboratories. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Bibliography

[ShaPhi] A. Shapiro and A. Philpott. *A Tutorial on Stochastic Programming*. 2007. (weblink)

Index