Open Source Tools for Optimization in Python

Ted Ralphs

COIN|OR

COMPUTATIONAL INFRASTRUCTURE FOR OPERATIONS RESEARCH

LEHIGH UNIVERSITY

COR@L

COMPUTATIONAL OPTIMIZATION RESEARCH AT LEHIGH

open source TM

SciPy 2015

IIT Bombay, 16 Decmber 2015

Outline

1. Introduction
2. PuLP
3. Pyomo
4. Solver Studio
5. Advanced Modeling
    - Sensitivity Analysis
    - Tradeoff Analysis (Multiobjective Optimization)
    - Nonlinear Modeling
    - Integer Programming
    - Stochastic Programming

Outline

1. Introduction
2. PuLP
3. Pyomo
4. Solver Studio
5. Advanced Modeling
   - Sensitivity Analysis
   - Tradeoff Analysis (Multiobjective Optimization)
   - Nonlinear Modeling
   - Integer Programming
   - Stochastic Programming

Algebraic Modeling Languages

- Generally speaking, we follow a four-step process in modeling.
  - Develop an abstract model.
  - Populate the model with data.
  - Solve the model.
  - Analyze the results.
- These four steps generally involve different pieces of software working in concert.
- For mathematical programs, the modeling is often done with an *algebraic modeling system*.
- Data can be obtained from a wide range of sources, including spreadsheets.
- Solution of the model is usually relegated to specialized software, depending on the type of model.

Open Source Solvers: COIN-OR

The COIN-OR Foundation

- A non-profit foundation promoting the development and use of interoperable, open-source software for operations research.
- A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.
- A venue for developing and maintaining standards.
- A forum for discussion and interaction between practitioners and researchers.

The COIN-OR Repository

- A collection of interoperable software tools for building optimization codes, as well as a few stand alone packages.
- A venue for peer review of OR software tools.
- A development platform for open source projects, including a wide range of project management tools.

The COIN-OR Optimization Suite

- COIN-OR distributes a free and open source suite of software that can handle all the classes of problems we'll discuss.
  - Clp (LP)
  - Cbc (MILP)
  - Ipopt (NLP)
  - SYMPHONY (MILP, BMILP)
  - DIP (MILP)
  - Bonmin (Convex MINLP)
  - Couenne (Non-convex MINLP)
  - Optimization Services (Interface)
- COIN also develops standards and interfaces that allow software components to interoperate.
- Check out the Web site for the project at `http://www.coin-or.org`

Installing the COIN-OR Optimization Suite

- Source builds out of the box on Windows, Linux, OSX using the Gnu autotools.
- Packages are available to install on many Linux distros, but there are some licensing issues.
- Homebrew recipes are available for many projects on OSX (we are working on this).
- For Windows, there is a GUI installer here:

  http://www.coin-or.org/download/binary/OptimizationSuite/

- For many more details, see Lecture 1 of this tutorial:

  http://coral.ie.lehigh.edu/ ted/teaching/coin-or

Modeling Software

Most existing modeling software can be used with COIN solvers.

- Commercial Systems
  - GAMS
  - MPL
  - AMPL
  - AIMMS
- Python-based Open Source Modeling Languages and Interfaces
  - Pyomo
  - PuLP/Dippy
  - CyLP (provides API-level interface)
  - yaposib

Modeling Software (cont'd)

- Other Front Ends (mostly open source)
  - FLOPC++ (algebraic modeling in C++)
  - CMPL
  - MathProg.jl (modeling language built in Julia)
  - GMPL (open-source AMPL clone)
  - ZMPL (stand-alone parser)
  - SolverStudio (spreadsheet plug-in: www.OpenSolver.org)
  - Open Office spreadsheet
  - R (RSymphony Plug-in)
  - Matlab (OPTI)
  - Mathematica

How They Interface

- Although not required, it's useful to know something about how modeling languages interface with solvers.
- In many cases, modeling languages interface with solvers by writing out an intermediate file that the solver then reads in.
- It is also possible to generate these intermediate files directly from a custom-developed code.
- Common file formats
  - MPS format: The original standard developed by IBM in the days of Fortran, not easily human-readable and only supports (integer) linear modeling.
  - LP format: Developed by CPLEX as a human-readable alternative to MPS.
  - .nl format: AMPL's intermediate format that also supports non-linear modeling.
  - OSIL: an open, XML-based format used by the Optimization Services framework of COIN-OR.
- Several projects use Python C Extensions to get the data into the solver through memory.

Where to Get the Examples

- The remainder of the talk will review a wide range of examples.
- These and many other examples of modeling with Python-based modeling languages can be found at the below URLs.

https://github.com/tkralphs/FinancialModels

http://projects.coin-or.org/browser/Dip/trunk/Dip/src/dippy/examples

https://github.com/Pyomo/PyomoGallery/wiki

https://github.com/coin-or/pulp/tree/master/examples
https://pythonhosted.org/PuLP/CaseStudies

Outline

1 Introduction

2 PuLP

3 Pyomo

4 Solver Studio

5 Advanced Modeling
- Sensitivity Analysis
- Tradeoff Analysis (Multiobjective Optimization)
- Nonlinear Modeling
- Integer Programming
- Stochastic Programming

PuLP: Algebraic Modeling in Python

- PuLP is a modeling language in COIN-OR that provides data types for Python that support algebraic modeling.
- PuLP only supports development of linear models.
- Main classes
  - `LpProblem`
  - `LpVariable`
- Variables can be declared individually or as "dictionaries" (variables indexed on another set).
- We do not need an explicit notion of a parameter or set here because Python provides data structures we can use.
- In PuLP, models are technically "concrete," since the model is always created with knowledge of the data.
- However, it is still possible to maintain a separation between model and data.

Simple PuLP Model (bonds_simple-PuLP.py)

```
from pulp import LpProblem, LpVariable, lpSum, LpMaximize, value

prob = LpProblem("Dedication Model", LpMaximize)

X1 = LpVariable("X1", 0, None)
X2 = LpVariable("X2", 0, None)

prob += 4*X1 + 3*X2
prob += X1 + X2 <= 100
prob += 2*X1 + X2 <= 150
prob += 3*X1 + 4*X2 <= 360

prob.solve()

print 'Optimal total cost is: ', value(prob.objective)

print "X1 :", X1.varValue
print "X2 :", X2.varValue
```

PuLP Model: Bond Portfolio Example (`bonds-PuLP.py`)

```
from pulp import LpProblem, LpVariable, lpSum, LpMaximize, value

from bonds import bonds, max_rating, max_maturity, max_cash

prob = LpProblem("Bond Selection Model", LpMaximize)

buy = LpVariable.dicts('bonds', bonds.keys(), 0, None)

prob += lpSum(bonds[b]['yield'] * buy[b] for b in bonds)

prob += lpSum(buy[b] for b in bonds) <= max_cash, "cash"

prob += (lpSum(bonds[b]['rating'] * buy[b] for b in bonds)
         <= max_cash*max_rating, "ratings")

prob += (lpSum(bonds[b]['maturity'] * buy[b] for b in bonds)
         <= max_cash*max_maturity, "maturities")
```

PuLP Data: Bond Portfolio Example (`bonds_data.py`)

```
bonds = {'A' : {'yield'     : 4,
                'rating'    : 2,
                'maturity'  : 3,},

         'B' : {'yield'     : 3,
                'rating'    : 1,
                'maturity'  : 4,},
        }

max_cash = 100
max_rating = 1.5
max_maturity = 3.6
```

Notes About the Model

- We can use Python's native `import` mechanism to get the data.
- Note, however, that the data is read and stored *before* the model.
- This means that we don't need to declare sets and parameters.
- Constraints
  - Naming of constraints is optional and only necessary for certain kinds of post-solution analysis.
  - Constraints are added to the model using an intuitive syntax.
  - Objectives are nothing more than expressions without a right hand side.
- Indexing
  - Indexing in Python is done using the native dictionary data structure.
  - Note the extensive use of comprehensions, which have a syntax very similar to quantifiers in a mathematical model.

Notes About the Data Import

- We are storing the data about the bonds in a "dictionary of dictionaries."
- With this data structure, we don't need to separately construct the list of bonds.
- We can access the list of bonds as `bonds.keys()`.
- Note, however, that we still end up hard-coding the list of features and we must repeat this list of features for every bond.
- We can avoid this using some advanced Python programming techniques, but how to do this with SolverStudio later.

Bond Portfolio Example: Solution in PuLP

```
prob.solve()

epsilon = .001

print 'Optimal purchases:'
for i in bonds:
    if buy[i].varValue > epsilon:
        print 'Bond', i, ":", buy[i].varValue
```

Example: Short Term Financing

A company needs to make provisions for the following cash flows over the coming five months: $-150K$, $-100K$, $200K$, $-200K$, $300K$.

- The following options for obtaining/using funds are available,
  - The company can borrow up to $\$100K$ at 1% interest per month,
  - The company can issue a 2-month zero-coupon bond yielding 2% interest over the two months,
  - Excess funds can be invested at 0.3% monthly interest.
- How should the company finance these cash flows if no payment obligations are to remain at the end of the period?

Example (cont.)

- All investments are risk-free, so there is no stochasticity.
- What are the decision variables?
  - x_i, the amount drawn from the line of credit in month i,
  - y_i, the number of bonds issued in month i,
  - z_i, the amount invested in month i,
- What is the goal?
  - To maximize the cash on hand at the end of the horizon.

Example (cont.)

The problem can then be modeled as the following linear program:

$$
\begin{aligned}
\max_{(x,y,z,v)\in\mathbb{R}^{12}} \quad & f(x,y,z,v) = v \\
\text{s.t.} \quad & x_1 + y_1 - z_1 = 150 \\
& x_2 - 1.01x_1 + y_2 - z_2 + 1.003z_1 = 100 \\
& x_3 - 1.01x_2 + y_3 - 1.02y_1 - z_3 + 1.003z_2 = -200 \\
& x_4 - 1.01x_3 - 1.02y_2 - z_4 + 1.003z_3 = 200 \\
& -1.01x_4 - 1.02y_3 - v + 1.003z_4 = -300 \\
& 100 - x_i \geq 0 \quad (i = 1, \ldots, 4) \\
& x_i \geq 0 \quad (i = 1, \ldots, 4) \\
& y_i \geq 0 \quad (i = 1, \ldots, 3) \\
& z_i \geq 0 \quad (i = 1, \ldots, 4) \\
& v \geq 0.
\end{aligned}
$$

PuLP Model for Short Term Financing (short_term_financing-PuLP.py)

```
from short_term_financing_data import cash, c_rate, b_yield
from short_term_financing_data import b_maturity, i_rate

T = len(cash)
credit = LpVariable.dicts("credit", range(-1, T), 0, None)
bonds = LpVariable.dicts("bonds", range(-b_maturity, T), 0, None)
invest = LpVariable.dicts("invest", range(-1, T), 0, None)

prob += invest[T-1]
for t in range(0, T):
    prob += (credit[t] - (1 + c_rate)* credit[t-1] +
            bonds[t] - (1 + b_yield) * bonds[t-int(b_maturity)] -
            invest[t] + (1 + i_rate) * invest[t-1] == cash[t])
prob += credit[-1] == 0
prob += credit[T-1] == 0
prob += invest[-1] == 0
for t in range(-int(b_maturity), 0):  prob += bonds[t] == 0
for t in range(T-int(b_maturity), T): prob += bonds[t] == 0
```

More Complexity: Facility Location Problem

- We have n locations and m customers to be served from those locations.
- There is a fixed cost c_j and a capacity W_j associated with facility j.
- There is a cost d_{ij} and demand w_{ij} for serving customer i from facility j.
- We have two sets of binary variables.
  - y_j is 1 if facility j is opened, 0 otherwise.
  - x_{ij} is 1 if customer i is served by facility j, 0 otherwise.

Capacitated Facility Location Problem

$$
\begin{array}{lll}
\min & \sum_{j=1}^{n} c_j y_j + \sum_{i=1}^{m} \sum_{j=1}^{n} d_{ij} x_{ij} & \\
\text{s.t.} & \sum_{j=1}^{n} x_{ij} = 1 & \forall i \\
& \sum_{i=1}^{m} w_{ij} x_{ij} \leq W_j & \forall j \\
& x_{ij} \leq y_j & \forall i, j \\
& x_{ij}, y_j \in \{0, 1\} & \forall i, j
\end{array}
$$

PuLP Model: Facility Location Example

```
from products   import REQUIREMENT, PRODUCTS
from facilities import FIXED_CHARGE, LOCATIONS, CAPACITY

prob = LpProblem("Facility_Location")

ASSIGNMENTS = [(i, j) for i in LOCATIONS for j in PRODUCTS]
assign_vars = LpVariable.dicts("x", ASSIGNMENTS, 0, 1, LpBinary)
use_vars    = LpVariable.dicts("y", LOCATIONS, 0, 1, LpBinary)

prob += lpSum(use_vars[i] * FIXED_COST[i] for i in LOCATIONS)

for j in PRODUCTS:
    prob += lpSum(assign_vars[(i, j)] for i in LOCATIONS) == 1

for i in LOCATIONS:
    prob += lpSum(assign_vars[(i, j)] * REQUIREMENT[j]
                  for j in PRODUCTS) <= CAPACITY * use_vars[i]

prob.solve()

for i in LOCATIONS:
    if use_vars[i].varValue > 0:
        print "Location ", i, " is assigned: ",
        print [j for j in PRODUCTS if assign_vars[(i, j)].varValue > 0]
```

PuLP Data: Facility Location Example

```
# The requirements for the products
REQUIREMENT = {
    1 : 7,
    2 : 5,
    3 : 3,
    4 : 2,
    5 : 2,
}
# Set of all products
PRODUCTS = REQUIREMENT.keys()
PRODUCTS.sort()
# Costs of the facilities
FIXED_COST = {
    1 : 10,
    2 : 20,
    3 : 16,
    4 : 1,
    5 : 2,
}
# Set of facilities
LOCATIONS = FIXED_COST.keys()
LOCATIONS.sort()
# The capacity of the facilities
CAPACITY = 8
```

Outline

1. Introduction
2. PuLP
3. Pyomo
4. Solver Studio
5. Advanced Modeling
   - Sensitivity Analysis
   - Tradeoff Analysis (Multiobjective Optimization)
   - Nonlinear Modeling
   - Integer Programming
   - Stochastic Programming

Pyomo

- An algebraic modeling language in Python similar to PuLP.
- Can import data from many sources, including AMPL-style data files.
- More powerful, includes support for nonlinear modeling.
- Allows development of both concrete models (like PuLP) and abstract models (like other AMLs).
- Also include PySP for stochastic Programming.
- Primary classes
  - `ConcreteModel`, `AbstractModel`
  - `Set`, `Parameter`
  - `Var`, `Constraint`
- Developers: Bill Hart, John Siirola, Jean-Paul Watson, David Woodruff, and others...

Example: Portfolio Dedication

- A pension fund faces liabilities totalling ℓ_j for years $j = 1, ..., T$.
- The fund wishes to dedicate these liabilities via a portfolio comprised of n different types of bonds.
- Bond type i costs c_i, matures in year m_i, and yields a yearly coupon payment of d_i up to maturity.
- The principal paid out at maturity for bond i is p_i.

LP Formulation for Portfolio Dedication

- We assume that for each year j there is at least one type of bond i with maturity $m_i = j$, and there are none with $m_i > T$.
- Let x_i be the number of bonds of type i purchased, and let z_j be the cash on hand at the beginning of year j for $j = 0, \ldots, T$. Then the dedication problem is the following LP.

$$
\begin{aligned}
\min_{(x,z)} \; & z_0 + \sum_i c_i x_i \\
\text{s.t. } & z_{j-1} - z_j + \sum_{\{i: m_i \geq j\}} d_i x_i + \sum_{\{i: m_i = j\}} p_i x_i = \ell_j, \quad (j = 1, \ldots, T-1) \\
& z_T + \sum_{\{i: m_i = T\}} (p_i + d_i) x_i = \ell_T. \\
& z_j \geq 0, j = 1, \ldots, T \\
& x_i \geq 0, i = 1, \ldots, n
\end{aligned}
$$

PuLP Model: Dedication (dedication-PuLP.py)

```
Bonds, Features, BondData, Liabilities = read_data('ded.dat')

prob = LpProblem("Dedication Model", LpMinimize)

buy  = LpVariable.dicts("buy", Bonds, 0, None)
cash = LpVariable.dicts("cash", range(len(Liabilities)), 0, None)

prob += cash[0] + lpSum(BondData[b, 'Price']*buy[b] for b in Bonds)

for t in range(1, len(Liabilities)):
    prob += (cash[t-1] - cash[t]
             + lpSum(BondData[b, 'Coupon'] * buy[b]
               for b in Bonds if BondData[b, 'Maturity'] >= t)
             + lpSum(BondData[b, 'Principal'] * buy[b]
               for b in Bonds if BondData[b, 'Maturity'] == t)
             == Liabilities[t], "cash_balance_%s"%t)
```

Notes on PuLP Model

- We are parsing the AMPL-style data file with a custom-written function `read_data` to obtain the data.
- The data is stored in a two-dimensional table (dictionary with tuples as keys).
- With Python supports of conditions in comprehensions, the model reads naturally in Python's native syntax.
- See also `FinancialModels.xlsx:Dedication-PuLP`.

Pyomo Model: Dedication (Concrete)

```
model = ConcreteModel()

Bonds, Features, BondData, Liabilities = read_data('ded.dat')

Periods = range(len(Liabilities))

model.buy = Var(Bonds, within=NonNegativeReals)
model.cash = Var(Periods, within=NonNegativeReals)
model.obj = Objective(expr=model.cash[0] +
            sum(BondData[b, 'Price']*model.buy[b] for b in Bonds),
            sense=minimize)

def cash_balance_rule(model, t):
    return (model.cash[t-1] - model.cash[t]
             + sum(BondData[b, 'Coupon'] * model.buy[b]
               for b in Bonds if BondData[b, 'Maturity'] >= t)
             + sum(BondData[b, 'Principal'] * model.buy[b]
               for b in Bonds if BondData[b, 'Maturity'] == t)
             == Liabilities[t])
model.cash_balance = Constraint(Periods[1:], rule=cash_balance_rule)
```

Notes on the Concrete Pyomo Model

- This model is almost identical to the PuLP model.
- The only substantial difference is the way in which constraints are defined, using "rules."
- Indexing is implemented by specifying additional arguments to the rule functions.
- When the rule function specifies an indexed set of constraints, the indices are passed through the arguments to the function.
- The model is constructed by looping over the index set, constructing each associated constraint.
- Note the use of the Python slice operator to extract a subset of a ranged set.

Instantiating and Solving a Pyomo Model

- The easiest way to solve a Pyomo Model is from the command line.

  ```
  pyomo -solver=cbc -summary dedication-PyomoConcrete.py
  ```

- It is instructive, however, to see what is going on under the hood.
  - Pyomo explicitly creates an "instance" in a solver-independent form.
  - The instance is then translated into a format that can be understood by the chosen solver.
  - After solution, the result is imported back into the instance class.
- We can explicitly invoke these steps in a script.
- This gives a bit more flexibility in post-solution analysis.

Instantiating and Solving a Pyomo Model

```
epsilon = .001

opt = SolverFactory("cbc")
instance = model.create()
results = opt.solve(instance)
instance.load(results)

print "Optimal strategy"
for b in instance.buy:
    if instance.buy[b].value > epsilon:
        print 'Buy %f of Bond %s' %(instance.buy[b].value,
                                     b)
```

Abstract Pyomo Model for Dedication (`dedication-PyomoAbstract.py`)

```
model = AbstractModel()

model.Periods = Set()
model.Bonds = Set()
model.Price = Param(model.Bonds)
model.Maturity = Param(model.Bonds)
model.Coupon = Param(model.Bonds)
model.Principal = Param(model.Bonds)
model.Liabilities = Param(range(9))

model.buy = Var(model.Bonds, within=NonNegativeReals)
model.cash = Var(range(9), within=NonNegativeReals)
```

Abstract Pyomo Model for Dedication (cont'd)

```
def objective_rule(model):
    return model.cash[0] + sum(model.Price[b]*model.buy[b]
                               for b in model.Bonds)
model.objective = Objective(sense=minimize, rulre=objective_rule)

def cash_balance_rule(model, t):
    return (model.cash[t-1] - model.cash[t]
             + sum(model.Coupon[b] * model.buy[b]
               for b in model.Bonds if model.Maturity[b] >= t)
             + sum(model.Principal[b] * model.buy[b]
               for b in model.Bonds if model.Maturity[b] == t)
             == model.Liabilities[t])

model.cash_balance = Constraint(range(1, 9),
                                rule=cash_balance_rule)
```

Notes on the Abstract Pyomo Model

- In an abstract model, we declare sets and parameters abstractly.
- After declaration, they can be used without instantiation, as in AMPL.
- When creating the instance, we explicitly pass the name of an AMPL-style data file, which is used to instantiate the concrete model.

```
instance = model.create('ded.dat')
```

- See also `FinancialModels.xlsx:Dedication-Pyomo`.

Outline

1 Introduction

2 PuLP

3 Pyomo

4 Solver Studio

5 Advanced Modeling
- Sensitivity Analysis
- Tradeoff Analysis (Multiobjective Optimization)
- Nonlinear Modeling
- Integer Programming
- Stochastic Programming

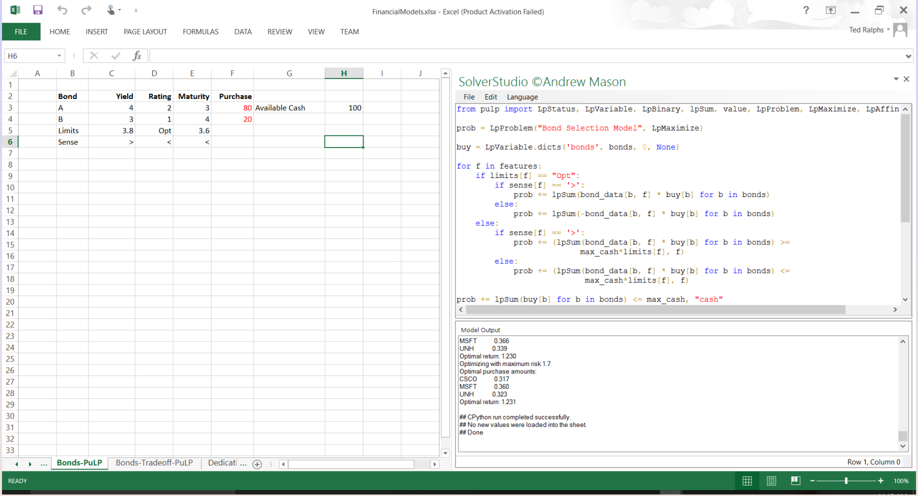
SolverStudio

- Spreadsheet optimization has had a (deservedly) bad reputation for many years.
- SolverStudio will change your mind about that!
- SolverStudio provides a full-blown modeling environment inside a spreadsheet.
  - Edit and run the model.
  - Populate the model from the spreadsheet.
- In many of the examples in the remainder of the talk, I will show the models in SolverStudio.

Bond Portfolio Example: PuLP Model in SolverStudio (FinancialModels.xlsx:Bonds-PuLP)

```
buy = LpVariable.dicts('bonds', bonds, 0, None)
for f in features:
    if limits[f] == "Opt":
        if sense[f] == '>':
            prob += lpSum(bond_data[b, f] * buy[b] for b in bonds)
        else:
            prob += lpSum(-bond_data[b, f] * buy[b] for b in bonds)
    else:
        if sense[f] == '>':
            prob += (lpSum(bond_data[b,f]*buy[b] for b in bonds) >=
                              max_cash*limits[f], f)
        else:
            prob += (lpSum(bond_data[b,f]*buy[b] for b in bonds) <=
                               max_cash*limits[f], f)
prob += lpSum(buy[b] for b in bonds) <= max_cash, "cash"
```

PuLP in Solver Studio



Notes About the SolverStudio PuLP Model

- We've explicitly allowed the option of optimizing over one of the features, while constraining the others.
- Later, we'll see how to create tradeoff curves showing the tradeoffs among the constraints imposed on various features.

Outline

1. Introduction
2. PuLP
3. Pyomo
4. Solver Studio
5. Advanced Modeling
    - Sensitivity Analysis
    - Tradeoff Analysis (Multiobjective Optimization)
    - Nonlinear Modeling
    - Integer Programming
    - Stochastic Programming

1 Introduction

2 PuLP

3 Pyomo

4 Solver Studio

5 Advanced Modeling
- Sensitivity Analysis
- Tradeoff Analysis (Multiobjective Optimization)
- Nonlinear Modeling
- Integer Programming
- Stochastic Programming

Marginal Price of Constraints

- The dual prices, or *marginal prices* allow us to put a value on "resources" (broadly construed).
- Alternatively, they allow us to consider the sensitivity of the optimal solution value to changes in the input.
- Consider the bond portfolio problem.
- By examining the dual variable for the each constraint, we can determine the value of an extra unit of the corresponding "resource".
- We can then determine the maximum amount we would be willing to pay to have a unit of that resource.
- The so-called "reduced costs" of the variables are the marginal prices associated with the bound constraints.

Sensitivity Analysis in PuLP and Pyomo

- Both PuLP and Pyomo also support sensitivity analysis through suffixes.
- Pyomo
  - The option `-solver-suffixes='.*'` should be used.
  - The supported suffixes are `.dual`, `.rc`, and `.slack`.
- PuLP
  - PuLP creates suffixes by default when supported by the solver.
  - The supported suffixed are `.pi` and `.rc`.

Sensitivity Analysis of the Dedication Model with PuLP

```
for t in Periods[1:]:
    prob += (cash[t-1] - cash[t]
             + lpSum(BondData[b, 'Coupon'] * buy[b]
               for b in Bonds if BondData[b, 'Maturity'] >= t)
             + lpSum(BondData[b, 'Principal'] * buy[b]
               for b in Bonds if BondData[b, 'Maturity'] == t)
             == Liabilities[t]), "cash_balance_%s"%t

status = prob.solve()

for t in Periods[1:]:
    print 'Present of $1 liability for period', t,
    print prob.constraints["cash_balance_%s"%t].pi
```

1 Introduction

2 PuLP

3 Pyomo

4 Solver Studio

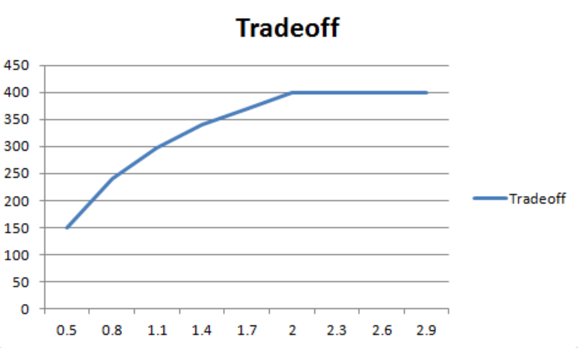
5 Advanced Modeling
- Sensitivity Analysis
- Tradeoff Analysis (Multiobjective Optimization)
- Nonlinear Modeling
- Integer Programming
- Stochastic Programming

Analysis with Multiple Objectives

- In many cases, we are trying to optimize multiple criteria simultaneously.
- These criteria often conflict (risk versus reward).
- Often, we deal with this by placing a constraint on one objective while optimizing the other.
- Extending the principles from the sensitivity analysis section, we can consider a doing a *parametric analysis*.
- We do this by varying the right-hand side systematically and determining how the objective function changes as a result.
- More generally, we ma want to find all *non-dominated* solutions with respect to two or more objectives functions.
- This latter analysis is called *multiobjective optimization*.

Parametric Analysis with PuLP (`FinancialModels.xlsx:Bonds-Tradeoff-PuLP`)

- Suppose we wish to analyze the tradeoff between yield and rating in our bond portfolio.
- By iteratively changing the value of the right-hand side of the constraint on the rating, we can create a graph of the tradeoff.



Parametric Analysis with PuLP

```
for r in range_vals:
    if sense[what_to_range] == '<':
        prob.constraints[what_to_range].constant = -max_cash*r
    else:
        prob.constraints[what_to_range].constant = max_cash*r

    status = prob.solve()

    epsilon = .001

    if LpStatus[status] == 'Optimal':
        obj_values[r] = value(prob.objective)
    else:
        print 'Problem is', LpStatus[status]
```

1 Introduction

2 PuLP

3 Pyomo

4 Solver Studio

5 Advanced Modeling
- Sensitivity Analysis
- Tradeoff Analysis (Multiobjective Optimization)
- Nonlinear Modeling
- Integer Programming
- Stochastic Programming

Portfolio Optimization

- An investor has a fixed amount of money to invest in a portfolio of n risky assets $S^1, \ldots, S^n$ and a risk-free asset S^0.
- We consider the portfolio's return over a fixed investment period $[0, 1]$.
- The random return of asset i over this period is

$$R_i := \frac{S_1^i}{S_0^i}.$$

- In general, we assume that the vector $\mu = \mathbb{E}[R]$ of expected returns is known.
- Likewise, $Q = \mathrm{Cov}(R)$, the variance-covariance matrix of the return vector R, is also assumed to be known.
- What proportion of wealth should the investor invest in asset i?

Trading Off Risk and Return

- To set up an optimization model, we must determine what our measure of "risk" will be.
- The goal is to analyze the tradeoff between risk and return.
- One approach is to set a target for one and then optimize the other.
- The classical portfolio model of Markowitz is based on using the variance of the portfolio return as a risk measure:

$$\sigma^2(R^\top x) = x^\top Qx,$$

  where $Q = \mathrm{Cov}(R_i, R_j)$ is the variance-covariance matrix of the vector of returns R.
- We consider three different single-objective models that can be used to analyze the tradeoff between these conflicting goals.

Markowitz Model

The Markowitz model is to maximize return subject to a limitation on the level of risk.

$$
\begin{array}{llr}
\text{(M2)} & \max_{x \in \mathbb{R}^n} & \mu^\top x \\
& \text{s.t.} & x^\top Q x \leq \sigma^2 \\
& & \displaystyle\sum_{i=1}^{n} x_i = 1,
\end{array}
$$

where σ^2 is the maximum risk the investor is willing to take.

Modeling Nonlinear Programs

- Pyomo support the inclusion of nonlinear functions in the model.
- A wide range of built-in functions are available.
- By restricting the form of the nonlinear functions, we ensure that the Hessian can be easily calculated.
- The solvers `ipopt`, `bonmin`, and `couenne` can be used to solve the models.
- See
  - `portfolio-*.mod`,
  - `portfolio-*-Pyomo.py`,
  - `FinancialModels.xlsx:Portfolio-AMPL`, and
  - `FinancialModels.xlsx:Portfolio-Pyomo`.

Pyomo Model for Portfolio Optimization (portfolio-Pyomo.py)

```
model = AbstractModel()

model.assets = Set()
model.T = Set(initialize = range(1994, 2014))
model.max_risk = Param(initialize = .00305)
model.R = Param(model.T, model.assets)
def mean_init(model, j):
    return sum(model.R[i, j] for i in model.T)/len(model.T)
model.mean = Param(model.assets, initialize = mean_init)
def Q_init(model, i, j):
    return sum((model.R[k, i] - model.mean[i])*(model.R[k, j]
                   - model.mean[j]) for k in model.T)
model.Q = Param(model.assets, model.assets, initialize = Q_init)

model.alloc = Var(model.assets, within=NonNegativeReals)
```

Pyomo model for Portfolio Optimization (cont'd)

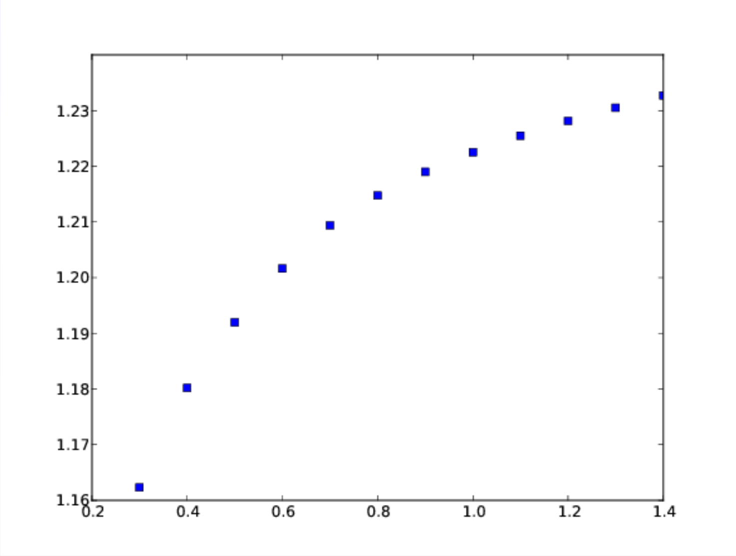
```
def risk_bound_rule(model):
    return (
        sum(sum(model.Q[i, j] * model.alloc[i] * model.alloc[j]
            for i in model.assets) for j in model.assets)
        <= model.max_risk)
model.risk_bound = Constraint(rule=risk_bound_rule)

def tot_mass_rule(model):
    return (sum(model.alloc[j] for j in model.assets) == 1)
model.tot_mass = Constraint(rule=tot_mass_rule)

def objective_rule(model):
    return sum(model.alloc[j]*model.mean[j] for j in model.assets)
model.objective = Objective(sense=maximize, rule=objective_rule)
```

Getting the Data

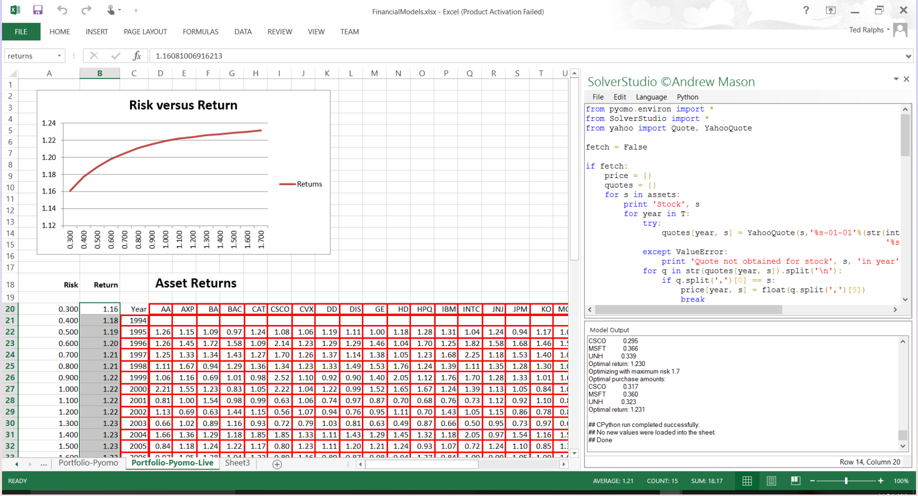
- One of the most compelling reasons to use Python for modeling is that there are a wealth of tools available.
- Historical stock data can be easily obtained from Yahoo using built-in Internet protocols.
- Here, we use a small Python package for getting Yahoo quotes to get the price of a set of stocks at the beginning of each year in a range.
- See `FinancialModels.xlsx:Portfolio-Pyomo-Live`.

```
for s in stocks:
    for year in range(1993, 2014):
        quote[year, s] = YahooQuote(s,'%s-01-01'%(str(year)),
                                      '%s-01-08'%(str(year)))
        price[year, s] = float(quote[year, s].split(',')[5])
            break
```

Efficient Frontier for the DJIA Data Set



Portfolio Optimization in SolverStudio



1. Introduction
2. PuLP
3. Pyomo
4. Solver Studio
5. Advanced Modeling
   - Sensitivity Analysis
   - Tradeoff Analysis (Multiobjective Optimization)
   - Nonlinear Modeling
   - **Integer Programming**
   - Stochastic Programming

Constructing an Index Fund

- An index is essentially a proxy for the entire universe of investments.
- An index fund is, in turn, a proxy for an index.
- A fundamental question is how to construct an index fund.
- It is not practical to simply invest in exactly the same basket of investments as the index tracks.
    - The portfolio will generally consist of a large number of assets with small associated positions.
    - Rebalancing costs may be prohibitive.
- A better approach may be to select a small subset of the entire universe of stocks that we predict will closely track the index.
- This is what index funds actually do in practice.

A Deterministic Model

- The model we now present attempts to cluster the stocks into groups that are "similar."
- Then one stock is chosen as the representative of each cluster.
- The input data consists of parameters ρ_{ij} that indicate the similarity of each pair (i, j) of stocks in the market.
- One could simply use the correlation coefficient as the similarity parameter, but there are also other possibilities.
- This approach is not guaranteed to produce an efficient portfolio, but should track the index, in principle.

An Integer Programming Model

- We have the following variables:
  - y_j is stock j is selected, 0 otherwise.
  - x_{ij} is 1 if stock i is in the cluster represented by stock j, 0 otherwise.
- The objective is to maximize the total similarity of all stocks to their representatives.
- We require that each stock be assigned to exactly one cluster and that the total number of clusters be q.

An Integer Programming Model

Putting it all together, we get the following formulation

$$
\begin{aligned}
\max \sum_{i=1}^{n} \sum_{j=1}^{n} \rho_{ij} x_{ij} & & & \quad (1) \\
\text{s.t.} \sum_{j=1}^{n} y_j = q & & & \quad (2) \\
\sum_{j=1}^{n} x_{ij} = 1 & & \forall i = 1, \dots, n & \quad (3) \\
x_{ij} \leq y_j & & \forall i = 1, \dots, n, j = 1, \dots, n & \quad (4) \\
x_{ij}, y_j \in \{0, 1\} & & \forall i = 1, \dots, n, j = 1, \dots, n & \quad (5)
\end{aligned}
$$

Constructing an Index Portfolio (IndexFund-Pyomo.py)

```
model.K = Param()
model.assets = Set()
model.T = Set(initialize = range(1994, 2014))
model.R = Param(model.T, model.assets)
def mean_init(model, j):
    return sum(model.R[i, j] for i in model.T)/len(model.T)
model.mean = Param(model.assets, initialize = mean_init)
def Q_init(model, i, j):
    return sum((model.R[k, i] - model.mean[i])*(model.R[k, j]
            - model.mean[j]) for k in model.T)
model.Q = Param(model.assets, model.assets, initialize = Q_init)

model.rep    = Var(model.assets, model.assets,
                   within=NonNegativeIntegers)
model.select = Var(model.assets,
                   within=NonNegativeIntegers)
```

Pyomo Model for Constructing an Index Portfolio (cont'd)

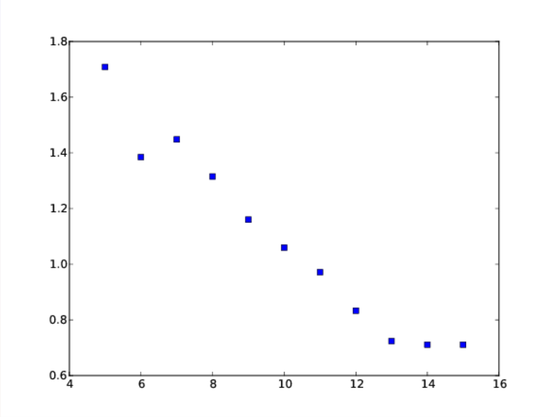
```
def representation_rule(model, i):
    return (sum(model.rep[i, j] for j in model.assets) == 1)
model.representation = Constraint(model.assets,
                                  rule=representation_rule)

def selection_rule(model, i, j):
    return (model.rep[i, j] <= model.select[j])
model.selection = Constraint(model.assets, model.assets,
                             rue=selection_rule)

def cardinality_rule(model):
    return (summation(model.select) == model.K)
model.cardinality = Constraint(rule=cardinality_rule)

def objective_rule(model):
    return sum(model.Q[i, j]*model.rep[i, j]
               for i in model.assets for j in model.assets)
model.objective = Objective(sense=maximize, ruke=objective_rule)
```

Interpreting the Solution

- As before, we let $\hat{w}$ be the relative market-capitalized weights of the selected stocks
$$\hat{w}_i = \frac{\sum_{j=1}^n z_i S^i x_{ij}}{\sum_{i=0}^n \sum_{j=1}^n z_i S^i x_{ij}},$$
where z_i is the number of shares of asset i that exist in the market and S^i the value of each share.
- This portfolio is what we now use to track the index.
- Note that we could also have weighted the objective by the market capitalization in the original model:
$$\max \sum_{i=1}^n \sum_{j=1}^n z_i S^i \rho_{ij} x_{ij}$$

Effect of K on Performance of Index Fund

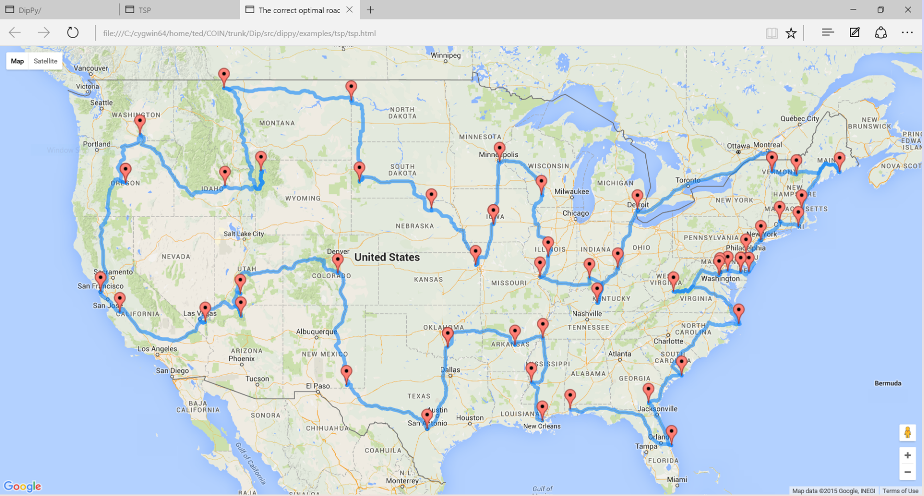
- This is a chart showing how the performance of the index changes as it's size is increased.
- This is for an equal-weighted index and the performance metric is sum of squared deviations.



Traveling Salesman Problem with Google Data

- In this next example, we develop a solver for the well-known TSP completely in Python.
- We obtain distance data using the Google Maps API.
- We solve the instance using Dippy (a Pulp derivative) and display the result back in Google Maps.

Traveling Salesman Problem with Google Data



1. Introduction
2. PuLP
3. Pyomo
4. Solver Studio
5. Advanced Modeling
   - Sensitivity Analysis
   - Tradeoff Analysis (Multiobjective Optimization)
   - Nonlinear Modeling
   - Integer Programming
   - Stochastic Programming

Building a Retirement Portfolio

- When I retire in 10 years or so :-), I would like to have a comfortable income.
- I'll need enough savings to generate the income I'll need to support my lavish lifestyle.
- One approach would be to simply formulate a mean-variance portfolio optimization problem, solve it, and then "buy and hold."
- This doesn't explicitly take into account the fact that I can periodically rebalance my portfolio.
- I may make a different investment decision today if I explicitly take into account that I will have *recourse* at a later point in time.
- This is the central idea of stochastic programming.

Modeling Assumptions

- In Y years, I would like to reach a savings goal of G.
- I will rebalance my portfolio every v periods, so that I need to have an investment plan for each of $T = Y/v$ periods (stages).
- We are given a universe $\mathcal{N} = \{1, \ldots, n\}$ of assets to invest in.
- Let $\mu_{it}, i \in \mathcal{N}, t \in \mathcal{T} = \{1, \ldots, T\}$ be the (mean) return of investment i in period t.
- For each dollar by which I exceed my goal of G, I get a reward of q.
- For each dollar I am short of G, I get a penalty of p.
- I have $\$B$ to invest initially.

Variables

- $x_{it}, i \in \mathcal{N}, t \in \mathcal{T}$: Amount of money to invest in asset i at beginning of period t t.
- z : Excess money at the end of horizon.
- w : Shortage in money at the end of the horizon.

A Naive Formulation

minimize

$$qz + pw$$

subject to

$$\begin{aligned}
\sum_{i \in \mathcal{N}} x_{i1} &= B \\
\sum_{i \in \mathcal{N}} x_{it} &= \sum_{i \in \mathcal{N}} (1 + \mu_{it}) x_{i,t-1} && \forall t \in \mathcal{T} \\
\sum_{i \in \mathcal{N}} (1 + \mu_{iT}) x_{iT} - z + w &= G \\
x_{it} &\geq 0 && \forall i \in \mathcal{N}, t \in \mathcal{T} \\
z, w &\geq 0
\end{aligned}$$

A Better Model

- What are some weaknesses of the model on the previous slide?
- Well, there are many...
- For one, it doesn't take into account the variability in returns (i.e., risk).
- Another is that it doesn't take into account my ability to rebalance my portfolio *after* observing returns from previous periods.
- I can and would change my portfolio after observing the market outcome.
- Let's use our standard notation for a market consisting of n assets with the price of asset i at the end of period t being denoted by the random variable S_t^i.
- Let $R_{it} = S_t^i / S_{t-1}^i$ be the return of asset i in period t.
- As we have done previously, let's take a scenario approach to specifying the distribution of R_{it}.

Scenarios

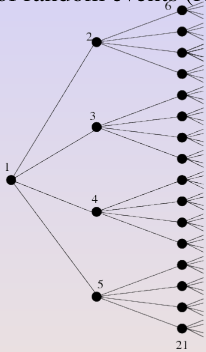
- We let the scenarios consist of all possible sequences of outcomes.
- Generally, we assume that for a particular realization of returns in period t, there will be M possible realizations for returns in period $t+1$.
- We then have M^T possible scenarios indexed by a set S.
- As before, we can then assume that we have a probability space (P^t, Ω^t) for each period t and that Ω^t is partitioned into $|S|$ subsets $\Omega^t_s, s \in S$.
- We then let $p^t_s = P(\Omega^t_s)\ \forall s \in S, t \in \mathcal{T}$.
- For instance, if $M = 4$ and $T = 3$, then we might have...

$t=1$	$t=2$	$t=3$
1	1	1
1	1	2
1	1	3
1	1	4
1	2	1
	$\vdots$	
4	4	4

- $|S| = 64$
- We can specify any probability on this outcome space that we would like.
- The time period outcomes don't need to be equally likely and returns in different time periods need not be mutually independent.

A Scenario Tree

- Essentially, we are approximating the continuous probability distribution of returns using a discrete set of outcomes.
- Conceptually, the sequence of random events (returns) can be arranged into a tree



Making it Stochastic

- Once we have a distribution on the returns, we could add uncertainty into our previous model simply by considering each scenario separately.
- The variables now become
  - $x_{its}, i \in \mathcal{N}, t \in \mathcal{T}$: Amount of money to reinvest in asset i at beginning of period t in scenario s.
  - $z_s, s \in S$: Excess money at the end of horizon in scenario s.
  - $w_s, s \in S$: Shortage in money at the end of the horizon in scenario s.
- Note that the return μ_{its} is now indexed by the scenario s.

A Stochastic Version: First Attempt

minimize

???????????????

subject to

$$
\begin{aligned}
\sum_{i \in \mathcal{N}} x_{i1} &= B \\
\sum_{i \in \mathcal{N}} x_{its} &= \sum_{i \in \mathcal{N}} (1 + \mu_{its}) x_{i,t-1,s} \qquad \forall t \in \mathcal{T}, \forall s \in S \\
\sum_{i \in \mathcal{N}} \mu_{iTs} x_{iTs} - z_s + w_s &= G \qquad \forall s \in S \\
x_{its} &\geq 0 \qquad \forall i \in \mathcal{N}, t \in \mathcal{T}, \forall s \in S \\
z_s, w_s &\geq 0 \qquad \forall s \in S
\end{aligned}
$$

Easy, Huh?

- We have just converted a multi-stage stochastic program into a deterministic model.
- However, there are some problems with our first attempt.
- What are they?

One Way to Fix It

- What we did to create our *deterministic equivalent* was to create copies of the variables for every scenario at every time period.
- One missing element is that we still have not have a notion of a probability distribution on the scenarios.
- But there's an even bigger problem...
- We need to enforce *nonanticipativity*...
- Let's define E_s^t as the set of scenarios with same outcomes as scenario s up to time t.
- At time t, the copies of all the anticipative decision variables corresponding to scenarios in E_s^t must have the same value.
- Otherwise, we will essentially be making decision at time t using information only available in periods after t.

A Stochastic Version: Explicit Nonanticipativity

minimize

$$\sum_{s\in S} p_s\left(qz_s - pw_s\right)$$

subject to

$$\begin{aligned}
\sum_{i\in\mathcal{N}} x_{i1} &= B \\
\sum_{i\in\mathcal{N}} x_{its} &= \sum_{i\in\mathcal{N}} (1+\mu_{its})x_{i,t-1,s} \qquad \forall t\in\mathcal{T}, \forall s\in S \\
\sum_{i\in\mathcal{N}} \mu_{iTs}x_{iTs} - z_s + w_s &= G \qquad \forall s\in S \\
x_{its} &= x_{its'}\ \forall i\in\mathcal{N}, \forall t\in T, \forall s\in S, \forall s'\in E_s^t \\
x_{its} &\geq 0 \qquad \forall i\in\mathcal{N}, t\in\mathcal{T}, \forall s\in S \\
z_s, w_s &\geq 0 \qquad \forall s\in S
\end{aligned}$$

Another Way

- We can also enforce nonanticipativity by using the “right” set of variables.
- We have a vector of variables for each node in the scenario tree.
- This vector corresponds to what our decision would be, given the realizations of the random variables we have seen so far.
- Index the nodes $= \{1, 2, \dots\}$.
- We will need to know the “parent” of any node.
- Let $A(l)$ be the ancestor of node $l \in$ in the scenario tree.
- Let $N(t)$ be the set of all nodes associated with decisions to be made at the beginning of period t.

Another Multistage Formulation

maximize

$$\sum_{l \in N(T)} p_l \left(q z_l + p w_l\right)$$

subject to

$$
\begin{aligned}
\sum_{i \in \mathcal{N}} x_{i1} &= B \\
\sum_{i \in \mathcal{N}} x_{il} &= \sum_{i \in \mathcal{N}} (1 + \mu_{il}) x_{i,A(l)} \qquad \forall l \in \\
\sum_{i \in \mathcal{N}} \mu_{il} x_{il} - z_l + w_l &= G \qquad \forall l \in N(T) \\
x_{il} &\geq 0 \qquad \forall i \in \mathcal{N}, l \in \\
z_l, w_l &\geq 0 \qquad \forall l \in N(T)
\end{aligned}
$$

PuLP Model for Retirement Portfolio (`DE-PuLP.py`)

```
Investments = ['Stocks', 'Bonds']

NumNodes = 21
NumScen = 64

b = 10000
G = 15000
q = 1 #0.05;
r = 2 #0.10;

Return = {
   0 : {'Stocks' : 1.25, 'Bonds' : 1.05},
    1 : {'Stocks' : 1.10, 'Bonds' : 1.05},
    2 : {'Stocks' : 1.00, 'Bonds' : 1.06},
    3 : {'Stocks' : 0.95, 'Bonds' : 1.08}
    }

NumOutcome = len(Return)
```

PuLP Model for Retirement Portfolio

```
x = LpVariable.dicts('x', [(i, j) for i in range(NumNodes)
                           for j in Investments], 0, None)
y = LpVariable.dicts('y', range(NumScen), 0, None)
w = LpVariable.dicts('w', range(NumScen), 0, None)

A  = dict([(k, (k-1)/NumOutcome) for k in range(1, NumNodes)])
A2 = dict([(s, 5 + s/NumOutcome) for s in range(NumScen)])
O  = dict([(k, (k-1) % NumOutcome)   for k in range(1, NumNodes)])
O2 = dict([(s, s % NumOutcome)   for s in range(NumScen)])

prob += lpSum(float(1)/NumScen * (q * y[s] + r * w[s])
              for s in range(NumScen))
prob += lpSum(x[0,i] for i in Investments) == b,
for k in range(1, NumNodes):
    prob += (lpSum(x[k,i] for i in Investments) ==
    lpSum(Return[O[k]][i] * x[A[k],i]  for i in Investments))
for s in range(NumScen):
    prob += lpSum(Return[O2[s]][i] * x[A2[s],i]
                  for i in Investments) - y[s] + w[s] == G
```

Thank You!