

*Exceptional
service
in the
national
interest*

0. Installing Pyomo



U.S. DEPARTMENT OF
ENERGY



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

- Install Anaconda3
 - <https://www.continuum.io/downloads>
 - **Install the Python 3.6 version** for your OS
 - Includes several packages for scientific computing already
 - Supports easy installation of pyomo and solvers
- Install pyomo and solvers:
 - Pyomo
 - Glpk (solver for linear and mixed-integer linear problems)
 - IPOPT (solver for continuous nonlinear problems)

Use Anaconda3 : Details [all platforms]



1. Download and Install Anaconda3 (<https://www.anaconda.com/download/>)

- “Python 3.6 version” for your platform.
- Large download (approx. 600 MB) including Python and other packages.
- Tips for the Mac: click on the .pkg file. You might get a message indicating that the package cannot be installed because it was downloaded from the internet. Try clicking again. If this does not work, navigate to the Downloads folder, right click (control-click) on Anaconda3-...x86_64.pkg and choose “Open With...” | “Installer.app”.

2. Test Python installation:

- On Windows, you will access Anaconda’s python from the “Anaconda Prompt” application. On the mac, you will use the terminal application. Both of these are command-line interfaces that allow you to run python code. Open “Anaconda Prompt” or a terminal app.
- Type “python --version”, and you should see something similar to:
Python 3.6.3 :: Anaconda, Inc.

Use Anaconda3 : Details [all platforms]



2. Test Python installation (con't):

- Type “python”. You should see something similar to:
Python 3.6.3 |Anaconda, Inc.| (default, Oct 6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
- Try writing some python code to test, e.g.,
>>> print('so.... this is python, eh?')
- Type “exit()” if everything is working correctly.

3. Install Pyomo:

- From the prompt, type: “conda install -c conda-forge pyomo”.
- Test your pyomo installation with “pyomo --version”. You should see something similar to the following:
Pyomo 5.3 (CPython 3.6.3 on Darwin 15.6.0)

Use Anaconda3 : Details [all platforms]



4. Install glpk:

- From the prompt, type: “conda install -c conda-forge glpk”
- Test the glpk solver. Type “glpsol --version”. You should see something like:
GLPSOL: GLPK LP/MIP Solver, v4.62
Copyright (C) 2000-2017 Andrew Makhorin, Department for Applied
...

5. Install IPOPT:

- From the prompt type: “conda install -c conda-forge ipopt”
- Test the ipopt solver. Type “ipopt --version”. You should see something like:
Ipopt 3.12.8 (Darwin x86_64), ASL(20160307)

Install with pip

[all platforms]  Sandia National Laboratories

- Install Python
 - Linux & Mac OS/X typically have Python pre-installed
 - Scientific Python distributions have many utilities pre-installed
 - <http://www.scipy.org/install.html>
- Install Pyomo
 - Install in your system
 - `pip install Pyomo`
 - (or) Install in a user directory
 - `pip install --user Pyomo`
- Install auxiliary software
 - Pyomo has conditional dependencies on various third-party packages
 - `pip install pyomo.extras`
 - Not needed for any of this tutorial
- Installing solvers
 - When using pip, solvers will need to be installed manually (or use the NEOS solver)

Getting help

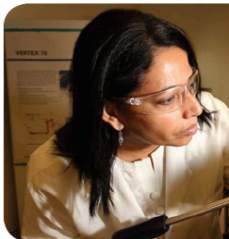
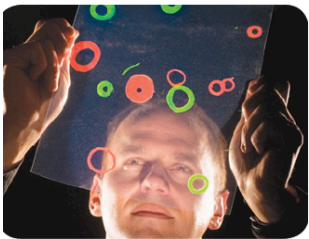
- Homepage:
 - www.pyomo.org

- GitHub site:
 - <https://github.com/Pyomo>

- Other sources
 - StackOverflow: “pyomo” tag
 - pyomo-forum@googlegroups.com
 - pyomo-developers@googlegroups.com

Acknowledgements

- William Hart
- Carl Laird
- John Siirola
- Jean-Paul Watson
- David Woodruff
- Bethany Nicholson
- Gabriel Hackebiel



*Exceptional
service
in the
national
interest*

Optional Material



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Install with Download Scripts [Windows]



- Install Python
 - Linux & Mac OS/X typically have Python pre-installed
 - Scientific Python distributions have many utilities pre-installed
 - <http://www.scipy.org/install.html>
- Install pip
 - <https://bootstrap.pypa.io/get-pip.py>
- Install Pyomo: the [get_pyomo.py](#) script
 - <https://software.sandia.gov/trac/pyomo/downloader>
- Install auxiliary software: the [get_pyomo_extras.py](#) script
 - <https://software.sandia.gov/trac/pyomo/downloader>

- Install Python
 - Linux & Mac OS/X typically have Python pre-installed
 - Scientific Python distributions have many utilities pre-installed
 - <http://www.scipy.org/install.htm>
- Install pyomo: the `pyomo_install` script
 - <https://software.sandia.gov/trac/pyomo/downloader>
 - Trunk install
 - `pyomo_install --trunk`
 - Install using a bundled zip file (no network access required)
 - `pyomo_install --zip=pyomo-zipfile.zip`
 - Install into a Virtual Python Environment
 - `pyomo_install --venv=pyomo`
 - *Trunk or zipfile options may be combined with virtual environments*

Install Solvers

Open Source Solvers

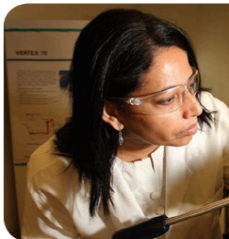
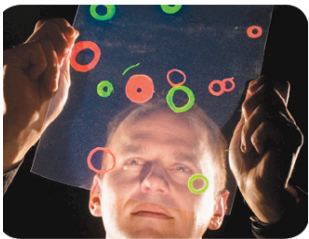
- COIN-OR Binary Distributions (e.g., CBC, IPOPT)
 - <http://www.coin-or.org/download/binary/>
 - Note: Coin-Binary requires the Coin-HSL Archive (Personal License)
 - <http://www.hsl.rl.ac.uk/ipopt/>
 - The Personal License permits commercial use ***but not redistribution***
- GLPK
 - <http://ftp.gnu.org/gnu/glpk/>
- SCIP
 - <http://scip.zib.de/#download>

Note: You need to add the solver installation to the PATH environment variable

License

Pyomo released under 3-clause BSD license

- No restrictions on deployment or commercial use



*Exceptional
service
in the
national
interest*

1. Overview of Pyomo



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Optimization Modeling

Goal:

- Provide a natural syntax to describe mathematical models
- Formulate large models with a concise syntax
- Separate modeling and data declarations
- Enable data import and export in commonly used formats

Impact:

- Robustly model large systems
- Integrated automatic differentiation for complex nonlinear models

Examples:

- AMPL, GAMS, AIMMS, ...
- OptimJ, FlopCPP, PuLP, JuMP, ...
- **Pyomo: A Python-based optimization modeling package**

Overview

- Three really good questions:
 - Why another Algebraic Modeling Language (AML)?
 - Why Python?
 - Why open-source?
- Pyomo: Team overview and collaborators / users
- Where to find more information...

Pyomo Overview

Idea: a Pythonic framework for formulating optimization models

- Keep a natural syntax even though it is Python
- Meet goals of modeling languages (concise, separate model and data)
- And...
- Provide an **open-source** tool for problem solving and research
- Allow for complex workflows through rich programming capabilities already available with Python

Highlights:

- Python provides a clean, intuitive syntax
- Python scripts provide a flexible context for exploring the structure of Pyomo models

```
# simple.py
from pyomo.environ import *

M = ConcreteModel()
M.x1 = Var()
M.x2 = Var(bounds=(-1,1))
M.x3 = Var(bounds=(1,2))
M.o = Objective(
    expr=M.x1**2 + (M.x2*M.x3)**4 + \
        M.x1*M.x3 + \
        M.x2*sin(M.x1+M.x3) + M.x2)
```

Pyomo Overview

- Equation-oriented models
 - User must provide *all* model equations explicitly (e.g., glass-box)
 - Not an optimizer around existing simulators (e.g., not black-box)
- Classes of problems and solvers:
 - Discrete Optimization (MIP): Gurobi, CPLEX, CBC, GLPK
 - Nonlinear Optimization (NLP): IPOPT, KNITRO, ... (Any AMPL or GAMS)
 - Mixed-Integer Nonlinear Optimization (MINLP): Baron, COUENNE
- Included packages and capabilities:
 - Pyomo Core: Basic modeling and optimization capabilities
 - Pyomo Blocks: Object-oriented (hierarchical) models
 - Pyomo.DAE: Modeling and optimization of dynamic systems (DAE and PDE)
 - Pyomo.GDP: Generalized disjunctive programming
 - PySP: Optimization under uncertainty
 - Pyomo.Bilevel, Pyomo.MPEC, ...

Pyomo Overview: Two focus areas

- **Solution Efficiency**

- Fast solution of large-scale problems (efficient interfaces with solvers)
- **HPC Ready**: Parallel execution of many optimization problems and parallel decomposition of large, structured problems

- **Flexibility (built on Python)**

- Ease of Prototyping and new workflows
- Higher-level interfaces and applications *through Python*
- Data import & manipulation, plotting, UI development

Why Model in Python?

- **Full-Featured Language**
 - Language features includes functions, classes, looping, namespaces, etc
 - Introspection facilitates the development of generic algorithms
 - Python's clean syntax facilitates rapid prototyping
- **Open Source License**
 - No licensing issues w.r.t. the language itself
- **Extensibility and Robustness**
 - Highly stable and well-supported
- **Support and Documentation**
 - Extensive online documentation and several excellent books
 - Long-term support for the language is not a factor
- **Standard Library**
 - Includes a large number of useful modules
- **Portability**
 - Widely available on many platforms

More than *just* mathematical modeling

Scripting

- Construct models using native Python data
- Iterative analysis of models leveraging Python functionality
- Data analysis and visualization of optimization results

Model transformations (a.k.a. reformulations)

- Automate generation of one model from another
- Leverage Pyomo's object model to apply transformations sequentially
- E.g.: DAE -> discretized model, GDP -> Big M

Meta-solvers

- Integrate scripting and/or transformations into optimization solver
- Leverage power of Python to build “generic” capabilities
- E.g.: progressive hedging, SP extensive form -> MIP

Why Open Source?

- Transparency and reliability
- Foster community involvement
 - Extend the modeling language
 - Develop new solvers / algorithms
 - Interface with additional external utilities
 - “Stone Soup” model
- Flexible licensing
 - Pyomo released under 3-clause BSD license
 - No restrictions on deployment or commercial use

Who Uses Pyomo?

- Students
 - Rose-Hulman, UC Davis, Purdue, U Texas, Iowa State, NPS, ...
- Researchers
 - Government laboratories
 - Sandia National Labs, Lawrence Livermore National Lab, Los Alamos National Lab, National Energy Technology Lab, Federal Energy Regulation Commission
 - Universities
 - UC Davis, TAMU, Rose-Hulman, UT, USC, GMU, Iowa State, NCSU, U Washington, NPS, U de Santiago de Chile, U Pisa, ...
 - Companies
- Software Projects
 - IDAES – Optimization of energy production systems
 - Minpower – Power systems toolkit
 - SolverStudio – Excel plugin for optimization modeling
 - TEMOA – Energy economy optimization models
 - Water Security Toolkit – Planning/Response for water contamination

For More Information

See the Pyomo homepage

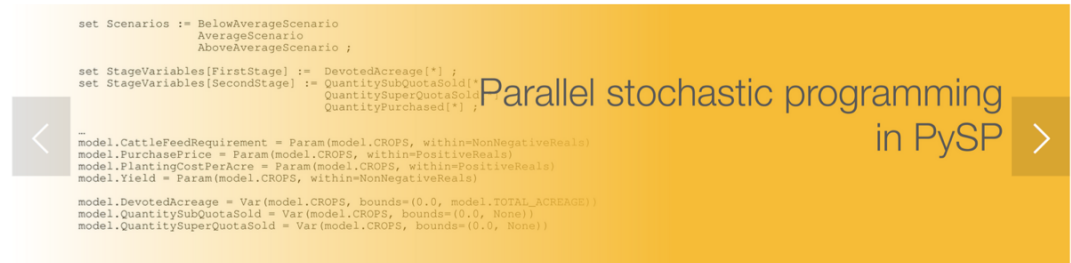
- www.pyomo.org

The Pyomo homepage provides a portal for:

- Online documentation
- Installation instructions
- Help information
- Developer links

GitHub:

- github.com/Pyomo



What Is Pyomo?

Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities.

[Read More](#)

Installation

The easiest way to install Pyomo is to use pip. Pyomo also needs access to optimization solvers.

[Read more](#)

Latest: Pyomo 4.4

Docs and Examples

Pyomo documentation and examples are available online. Pyomo is also described in book and journal publications.

[Read more](#)

Acknowledgments

The Pyomo project would not be where it is without the generous contributions of numerous people and organizations.

[Read More](#)

Getting Help

Users and developers provide help online:

- [Questions on StackExchange](#)
- [Discussions on Pyomo Forum](#)

Who Uses Pyomo?

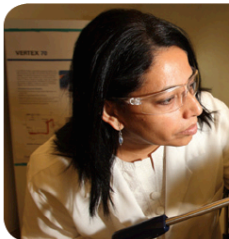
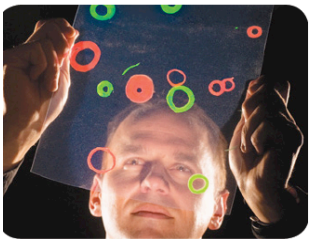
Pyomo is used by researchers to solve complex real-world applications.

[Read More](#)

ASK A QUESTION

Acknowledgements

- William Hart (SNL)
- Jean-Paul Watson (SNL)
- John Siirola (SNL)
- Francisco Munoz (SNL, UAI)
- Bethany Nicholson (CMU, SNL)
- Prof. David L. Woodruff (UC Davis)
- Prof. Roger Wets (UC Davis)
- Prof. Carl D. Laird (Purdue, SNL)
- Gabe Hackebeil (U.Mich.)



Sandia
National
Laboratories

*Exceptional
service
in the
national
interest*

2. A Python Tutorial



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Why Python?

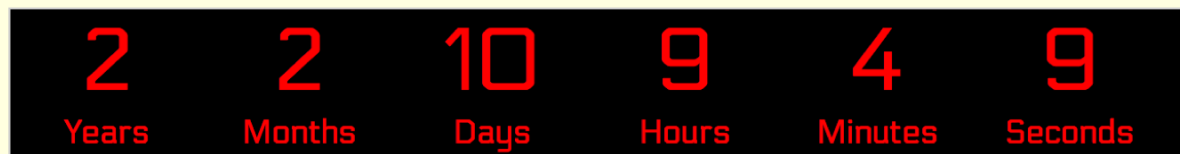
- Interpreted language
- Intuitive syntax
- Object-oriented
- Simple, but extremely powerful
- Lots of built-in libraries and third-party extensions
- Shallow learning curve and many resources
- Dynamic typing
- Integration with C/Java

Python Versions: 2.x vs 3.x

- Python 3.0 was released in 2008
 - Included significant backward incompatibilities
 - Initial adoption was slow
 - But, community is moving on to Python 3.0
- Pyomo support:
 - Python 2.7
 - Very stable; patches have included package updates to support Python 3.x compatibility
 - Python 3.5, 3.6
 - Very stable

*We try to stick to
“universal” syntax
that will work in
both 2.x and 3.x*

Python 2.7 will retire in...



<https://pythonclock.org>

Overview

- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules
- exceptions
- files & standard library

Interactive Shell

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations:
 - IDLE (GUI)
 - python (command line)
- Type statements or expressions at prompt:

```
>>> print( "Hello, world" )
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

Python Program

- To write a program, put commands in a file

```
# hello.py  
print( "Hello, world" )  
x = 12**2  
print( x )
```

- Execute on the command line

```
C:\Users\me> python hello.py  
Hello, world  
144
```

Python Variables

- No need to declare
- Need to assign (initialize)
 - use of uninitialized variable raises exception

- Not typed

```
greeting = 34.2
if friendly:
    greeting = "hello world"
else:
    greeting = 12**2
print( greeting )
```

- ***Everything*** is a "variable":
 - Even functions, classes, modules

Control Structures

```
if condition:  
    statements
```

```
elif condition:  
    statements ...
```

```
else:  
    statements
```

```
while condition:  
    statements
```

```
for var in sequence:  
    statements
```

```
break  
continue
```

Note: *Spacing matters!*

Control structure scope dictated by indentation

Grouping Indentation

In Python:

```
for i in range(20):  
    if i % 3 == 0:  
        print(i)  
        if i % 5 == 0:  
            print("Bingo!")  
    print("----")
```

In C:

```
for (i = 0; i < 20; i++)  
{  
    if (i % 3 == 0) {  
        printf("%d\n", i);  
        if (i % 5 == 0) {  
            printf("Bingo!\n");  
        }  
    }  
    printf("----\n");  
}
```


Numbers

- The usual suspects
 - 12, 3.14, 0xFF, 0377, (-1+2)*3/4**5, abs(x), 0<x<=5
- C-style shifting & masking
 - 1<<16, x&0xff, x|1, ~x, x^y
- Integer division truncates
 - Python 2.x
 - 1/2 → 0, 1./2. → 0.5, float(1)/2 → 0.5
 - from __future__ import division
 - » 1/2 → 0.5
 - Python 3.x
 - 1/2 → 0.5
- Long (arbitrary precision), complex
 - 2L**100 → 1267650600228229401496703205376L
 - In Python 2.2 and beyond, 2**100 does the same thing
 - 1j**2 → (-1+0j)

Strings

- `"hello"+"world"` `"helloworld"` *# concatenation*
 - `"hello"*3` `"hellohellohello"` *# repetition*
 - `"hello"[0]` `"h"` *# indexing*
 - `"hello"[-1]` `"o"` *# (from end)*
 - `"hello"[1:4]` `"ell"` *# slicing*
 - `len("hello")` `5` *# size*
 - `"hello" < "jello"` `True` *# comparison*
 - `"e" in "hello"` `True` *# search*
-
- `"escapes: \n etc, \033 etc, \if etc"`
 - `'single quotes'    ""triple quotes""    r"raw strings"`

Lists

- Flexible arrays, *not* linked lists
 - `a = [99, "bottles of beer", ["on", "the", "wall"]]`
- Same operators as for strings
 - `a+b, a*3, a[0], a[-1], a[1:], len(a)`
- Item and slice assignment
 - `a[0] = 98`
 - `a[1:2] = ["bottles", "of", "beer"]`
`# -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]`
 - `del a[-1]`
`# -> [98, "bottles", "of", "beer"]`

List Operations

```
>>> a = range(5)           # [0,1,2,3,4]
>>> a.append(5)            # [0,1,2,3,4,5]
>>> a.pop()                # [0,1,2,3,4]
5
>>> a.insert(0, 42)        # [42,0,1,2,3,4]
>>> a.pop(0)               # [0,1,2,3,4]
42
>>> a.reverse()           # [4,3,2,1,0]
>>> a.sort()               # [0,1,2,3,4]
```

Dictionaries

- Hash tables, "associative arrays"
 - `d = {"duck": "eend", "water": "water"}`
- Lookup:
 - `d["duck"]` *# -> "eend"*
 - `d["back"]` *# raises KeyError exception*
- Delete, insert, overwrite:
 - `del d["water"]` *# {"duck": "eend", "back": "rug"}*
 - `d["back"] = "rug"` *# {"duck": "eend", "back": "rug"}*
 - `d["duck"] = "duik"` *# {"duck": "duik", "back": "rug"}*

Dictionary Operations

- Keys, values, items:

- `d.keys()` $\rightarrow$ `["duck", "back"]`
- `d.values()` $\rightarrow$ `["duik", "rug"]`
- `d.items()` $\rightarrow$ `[("duck", "duik"), ("back", "rug")]`

Note: These actually return generators, not lists.

- Presence check:

- `d.has_key("duck")` *# $\rightarrow$ 1; `d.has_key("spam")` $\rightarrow$ 0*

- Values of any type; keys almost any

- ```
{ "name": "Guido",
 "age": 43,
 ("hello", "world"): 1,
 42: "yes",
 "flag": ["red", "white", "blue"] }
```

# Dictionary Details

---

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - keys are hashed (to ensure fast lookup)
  - lists or dictionaries cannot be used as keys
    - these objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - key hash values are in an arbitrary order
  - that numeric keys are returned sorted is an artifact of the implementation and *is not guaranteed*

# References vs. Copies

---

- Assignment manipulates references
  - `x = y` **does not make a copy** of `y`
  - `x = y` makes `x` **reference** the object `y` references

- Reference values can be modified!

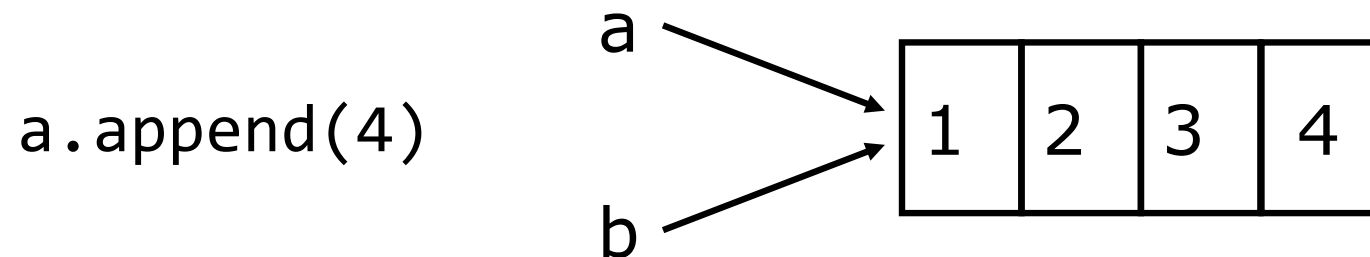
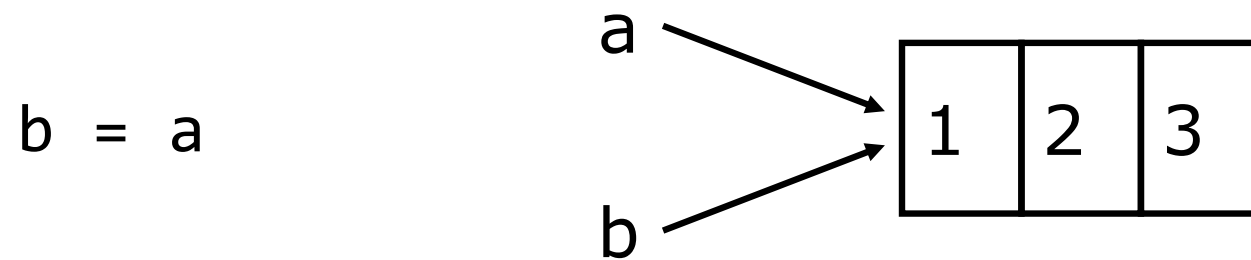
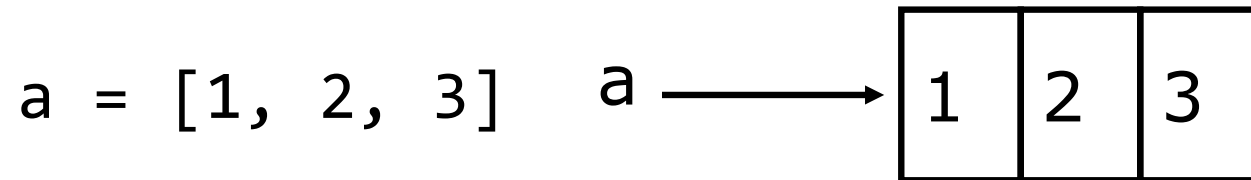
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print(b)
[1, 2, 3, 4]
```

- Copied objects are distinct

```
>>> import copy
>>> c = copy.copy(a)
>>> a.pop()
>>> print(c)
[1, 2, 3, 4]
```

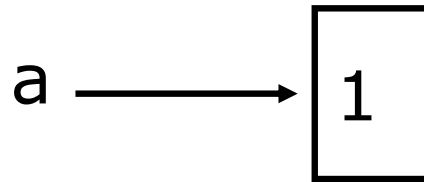


# Changing a Shared List

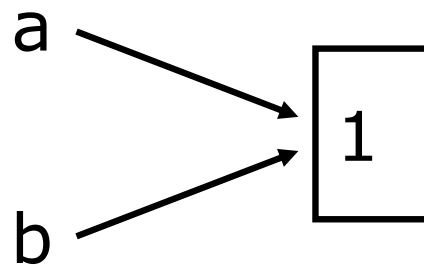


# Changing an Integer

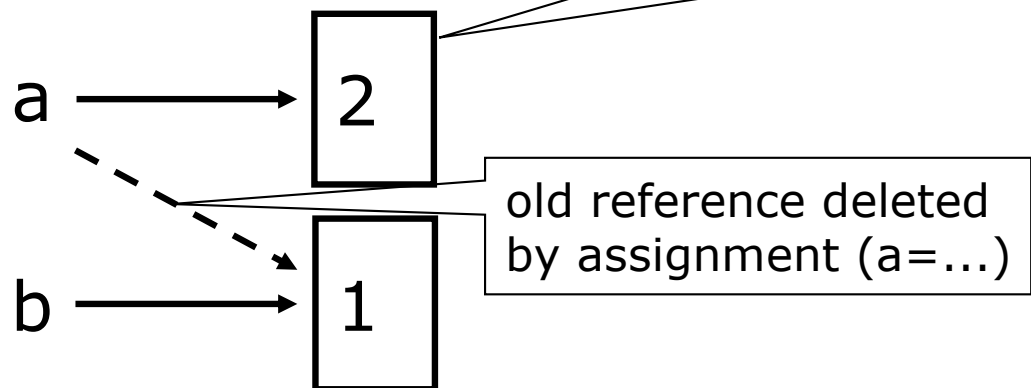
`a = 1`



`b = a`



`a = a+1`



# Functions / Procedures

---

```
def name(arg1, arg2, ...):
 """documentation""" # optional doc string
 statements

 return expression # from function
 return # from procedure (returns None)
```

# Example

---

```
def gcd(a, b):
 """greatest common divisor"""
 while a != 0:
 a, b = b%a, a # parallel assignment
 return b
```

```
>>> gcd.__doc__
'greatest common divisor'
```

```
>>> gcd(12, 20)
4
```

# Modules

---

- Collection of stuff in *foo.py* file
  - functions, classes, variables

- Importing modules:

```
import re
print(re.match("[a-z]+", s))
from re import match
print(match("[a-z]+", s))
```

- Import with rename:

```
import re as regex
from re import match as m
```

# Major Python Packages

---

- Matplotlib
  - 2D plotting library
  - <http://matplotlib.org/>
- Pandas
  - Data structures and analysis
  - <http://pandas.pydata.org/>
- NetworkX, Plotly, ...
- SciPy
  - Scientific Python for mathematics and engineering
  - <http://www.scipy.org>
- Numpy
  - Numeric array package
  - <http://www.numpy.org/>
- Ipython
  - Interactive Python shell
  - <http://ipython.org/>

# Resources

---

- Software Carpentry
  - <http://software-carpentry.org/>
- Python webpage
  - <http://www.python.org>
- Books
  - Python Essential Reference (4th Edition), David Beazley, 2009
  - Python in a Nutshell, Alex Martelli, 2003
  - Python Pocket Reference, 4th Edition, Mark Lutz, 2009
  - ...

# Acknowledgements

---

- William Hart
- Ted Ralphs
- John Siirola
- Carl Laird
- Bethany Nicholson
- Dave Woodruff
- Guido van Rossum



# OTHER MATERIAL

# Python Implementations

- Cpython
    - C Python interpreter
    - <https://www.python.org/downloads/>
    - SciPy Stack
      - <http://www.scipy.org/install.html>
      - Anaconda: Linux/MacOS/MS Windows
  - PyPy
    - A Python interpreter written in Python
    - <http://pypy.org/>
  - Jython
    - Java Python interpreter
    - <http://www.jython.org/>
  - IronPython
    - .NET Python interpreter
    - <http://ironpython.net/>
- Full Pyomo Support
- Beta Pyomo Support
- Pyomo Not Supported (yet)

# Tuples

---

- `key = (lastname, firstname)`
- `point = x, y, z` *# parentheses optional*
- `x, y, z = point` *# unpack*
- `lastname = key[0]` *# index tuple values*
- `singleton = (1,)` *# trailing comma!!!*  
*# (1) → integer!*
- `empty = ()` *# parentheses!*
  
- Tuples vs. lists
  - tuples immutable
  - lists mutable

# Classes

---

```
class name(object):
 """documentation"""
 statements
```

Most, *statements* are method definitions:

```
def name(self, arg1, arg2, ...):
 ...
```

May also be *class variable* assignments

# Example

---

```
class Stack(object):
 """A well-known data structure..."""

 def __init__(self): # constructor
 self.items = []

 def push(self, x): # the sky is the limit
 self.items.append(x)

 def pop(self): # what if it's empty?
 x = self.items[-1]
 del self.items[-1]
 return x

 def empty(self):
 return len(self.items) == 0 # Boolean result
```

# Example (cont'd)

---

- To create an instance, simply call the class object:

```
x = Stack() # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty() # -> 1
x.push(1) # [1]
x.empty() # -> 0
x.push("hello") # [1, "hello"]
x.pop() # -> "hello" # [1]
```

- To inspect instance variables, use dot notation:

```
x.items # -> [1]
```

# Class/Instance Variables

---

```
class Connection(object):
 verbose = 0 # class variable

 def __init__(self, host):
 self.host = host # instance variable

 def debug(self, v):
 self.verbose = v # make instance variable!

 def connect(self):
 if self.verbose: # class or instance variable?
 print("connecting to %s" % (self.host,))
```

# Instance Variable Rules

---

- On use via instance (`self.x`), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (`self.x = ...`):
  - always makes an instance variable
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own



# Catching Exceptions

---

```
def foo(x):
 return 1/x

def bar(x):
 try:
 print(foo(x))
 except ZeroDivisionError as message:
 print("Can't divide by zero: %s" % message)

bar(0)
```

# Try-Finally: Cleanup

---

```
f = open(file)
try:
 process_file(f)
finally:
 f.close()
print("OK")
```

*# always executed*  
*# executed on success only*

# Raising Exceptions

---

```
raise IndexError
```

```
raise IndexError("k out of range")
```

```
raise IndexError, "k out of range"
 # this only works in Python 2.x!
```

```
try:
 something
except: # catch everything
 print("Oops")
 raise # reraise
```

# More on Exceptions

---

- User-defined exceptions
  - subclass `Exception` or any other standard exception
- Note: in older versions of Python exceptions can be strings
- Last caught exception info:
  - `sys.exc_info() == (exc_type, exc_value, exc_traceback)`
- Printing exceptions: `traceback` module

# File Objects

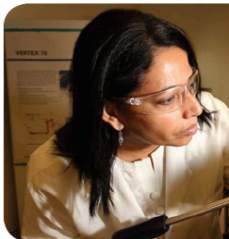
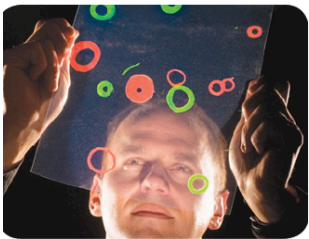
---

- `f = open(filename[, mode[, buffersize]])`
  - mode can be "r", "w", "a" (like C stdio); default "r"
  - append "b" for text translation mode
  - append "+" for read/write open
  - buffersize: 0=unbuffered; 1=line-buffered; buffered
  
- methods:
  - `read([nbytes])`, `readline()`, `readlines()`
  - `write(string)`, `writelines(list)`
  - `seek(pos[, how])`, `tell()`
  - `flush()`, `close()`
  - `fileno()`

# Highlights from the Standard Library

---

- Core:
  - os, sys, string, getopt, StringIO, struct, pickle, json, csv, collections, math, tempfile, itertools, ...
- Regular expressions:
  - re module; Perl-5 style patterns and matching rules
- Internet:
  - socket, rfc822, httpplib, htmlplib, ftplib, smtpplib, ...
- Miscellaneous:
  - pdb (debugger), profile+pstats
  - Tkinter (Tcl/Tk interface), audio, \*dbm, ...



*Exceptional  
service  
in the  
national  
interest*

### 3. Pyomo Fundamentals



Sandia National Laboratories is a multission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Learning Objectives

---

- Goals, benefits, drawbacks of Pyomo
- Basic understanding of Python
- Create and solve optimization problems within Pyomo
  - Vars, Constraints, Objectives
  - Sets, Parameters
- Basic scripting, workflow, and programming capabilities
  - Changing data, multiple solutions, importing data, plotting, reporting
  - Pieces to build your own workflows



# Simple Modeling Example: Classic Knapsack Problem

- Given a set of items  $A$ , with weight and value (benefit)
- Goal: select a subset of these items
  - Maximize the benefit
  - Under weight limit

| Item (A)    | Weight (w) | Benefit (b) |
|-------------|------------|-------------|
| hammer      | 5          | 8           |
| wrench      | 7          | 3           |
| screwdriver | 4          | 6           |
| towel       | 3          | 11          |

Max weight: 14

| Symbol     | Meaning                               |
|------------|---------------------------------------|
| $A$        | set of items available for purchase   |
| $b_i$      | benefit of item $i$                   |
| $w_i$      | weight of item $i$                    |
| $W_{\max}$ | max weight that can be carried        |
| $x_i$      | discrete variable (select $i$ or not) |

# Simple Modeling Example: Classic Knapsack Problem

- Given a set of items  $A$ , with weight and value (benefit)
- Goal: select a subset of these items
  - Maximize the benefit
  - Under weight limit

| Item (A)    | Weight (w) | Benefit (b) |
|-------------|------------|-------------|
| hammer      | 5          | 8           |
| wrench      | 7          | 3           |
| screwdriver | 4          | 6           |
| towel       | 3          | 11          |

Max weight: 14

| Symbol     | Meaning                               |
|------------|---------------------------------------|
| $A$        | set of items available for purchase   |
| $b_i$      | benefit of item $i$                   |
| $w_i$      | weight of item $i$                    |
| $W_{\max}$ | max weight that can be carried        |
| $x_i$      | discrete variable (select $i$ or not) |

$$\begin{aligned}
 & \max_x \quad \sum_{i \in A} b_i x_i \\
 & s. t. \quad \sum_{i \in A} w_i x_i \leq W_{\max} \\
 & \quad \quad x_i \in \{0,1\} \quad \forall i \in A
 \end{aligned}$$

# Simple Modeling Example: Classic Knapsack Problem

- Variables
- Objectives
- Constraints
- Sets
- Parameters

| Item (A)    | Weight (w) | Benefit (b) |
|-------------|------------|-------------|
| hammer      | 5          | 8           |
| wrench      | 7          | 3           |
| screwdriver | 4          | 6           |
| towel       | 3          | 11          |

Max weight: 14

| Symbol     | Meaning                               |
|------------|---------------------------------------|
| $A$        | set of items available for purchase   |
| $b_i$      | benefit of item $i$                   |
| $w_i$      | weight of item $i$                    |
| $W_{\max}$ | max weight that can be carried        |
| $x_i$      | discrete variable (select $i$ or not) |

$$\begin{aligned}
 & \max_x \quad \sum_{i \in A} b_i x_i \\
 & s. t. \quad \sum_{i \in A} w_i x_i \leq W_{\max} \\
 & \quad \quad x_i \in \{0,1\} \quad \forall i \in A
 \end{aligned}$$

# Anatomy of a Concrete Pyomo Model

$$\begin{aligned} \max_x \quad & \sum_{i \in A} v_i x_i \\ \text{s.t.} \quad & \sum_{i \in A} w_i x_i \leq W_{\max} \\ & x_i \in \{0,1\} \quad \forall i \in A \end{aligned}$$

| Item        | Weight | Value |
|-------------|--------|-------|
| hammer      | 5      | 8     |
| wrench      | 7      | 3     |
| screwdriver | 4      | 6     |
| towel       | 3      | 11    |

```

from pyomo.environ import *

A = ['hammer', 'wrench', 'screwdriver', 'towel']
v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
W_max = 14

model = ConcreteModel()
model.x = Var(A, within=Binary)

model.value = Objective(
 expr = sum(v[i]*model.x[i] for i in A),
 sense = maximize)

model.weight = Constraint(
 expr = sum(w[i]*model.x[i] for i in A) <= W_max)

opt = SolverFactory('glpk')
result_obj = opt.solve(model, tee=True)

model.pprint()

```

← Import packages

← Specify/import data

Can be imported or even  
applied to model later

← Define problem

← Solve problem

← Report results

# Pyomo imports and *namespaces*

- Pyomo objects exist within the `pyomo.environ` namespace:

```
import pyomo.environ
model = pyomo.environ.ConcreteModel()
```

- ...but this gets verbose. To save typing, we will import the core Pyomo classes into the main namespace:

```
from pyomo.environ import *
model = ConcreteModel()
```

- To clarify Pyomo-specific syntax in this tutorial, we will highlight Pyomo symbols in **green**

- Note: We violate good Python practice in this tutorial. One should:

```
import pyomo.environ as pe
model = pe.ConcreteModel()
```

# Getting Started: the *Model*

```
from pyomo.environ import *
```

```
model = ConcreteModel()
```

← Every Pyomo model starts with this; it tells Python to load the Pyomo Modeling Environment

↑ Create an instance of a *Concrete* model

- Concrete models are immediately constructed
- Data must be present *at the time* components are defined

↑ Local variable to hold the model we are about to construct

- While not required, by convention we use “model”

# Populating the Model: *Variables*

Components can take a variety of keyword arguments when constructed.

`model.a_variable = Var(within = NonNegativeReals)`

The name you assign the object to becomes the object's name, and must be unique in any given model.

"within" is optional and sets the variable domain ("domain" is an alias for "within")

Several pre-defined domains, e.g., "Binary"

# Populating the Model: *Variables*

Components can take a variety of keyword arguments when constructed.

`model.a_variable = Var(within = NonNegativeReals)`

The name you assign the object to becomes the object's name, and must be unique in any given model.

"within" is optional and sets the variable domain ("domain" is an alias for "within")

Several pre-defined domains, e.g., "Binary"

`model.a_variable = Var(bounds = (0, None))`

`model.a_variable = Var(initialize = 42.0)`

`model.a_variable = Var(initialize=42.0, bounds=(0, None))`



# Defining the *Objective*

```
model.x = Var(initialize=-1.2, bounds=(-2, 2))
model.y = Var(initialize= 1.0, bounds=(-2, 2))
```

```
model.obj = Objective(
 expr= (1-model.x)**2 + 100*(model.y-model.x**2)**2,
 sense= minimize)
```

If “sense” is omitted, Pyomo assumes minimization

“expr” can be an expression, or any function-like object that returns an expression

Note that the Objective expression is not a *relational expression*

# Defining the Problem: *Constraints*

```
model.a = Var()
model.b = Var()
model.c = Var()
model.c1 = Constraint(
 expr = model.b + 5 * model.c <= model.a)
```

$$b + 5c \leq a$$

↑  
“expr” can be an expression,  
or any function-like object  
that returns an expression

```
model.c2 = Constraint(expr = (None, model.a + model.b, 1))
```

↑  
“expr” can also be a tuple:

- 3-tuple specifies ( LB, expr, UB )
- 2-tuple specifies an equality constraint.

$$a + b \leq 1$$

# Higher-dimensional components

- (Almost) All Pyomo *components* can be *indexed*

```
A = [1, 2, 5]
```

```
model.x = Var(A)
```

- All non-keyword arguments are assumed to be *indices*
- Individual indices may be multi-dimensional (e.g., a list of pairs)

- E.g., Indexed variables

```
A = [1, 2, 5]
```

```
B = ['wrench', 'hammer']
```

```
model.x = Var(A)
```

```
model.y = Var(A, B)
```

The indexes are any iterable object,  
e.g., list or Set



- **Note:** while indexed variables look like matrices, they are **not**.
  - In particular, we do not support matrix algebra (*yet...*)

# List Comprehensions

```
model.IDX = range(10)
model.a = Var()
model.b = Var(model.IDX)
model.c1 = Constraint(
 expr = sum(model.b[i] for i in model.IDX) <= model.a)
```

Python *list comprehensions* are very common for working over indexed variables and nicely parallel mathematical notation:

$$\sum_{i \in \text{IDX}} b_i \leq a$$

# Putting it all together: Concrete Knapsack

```
knapsack.py
from pyomo.environ import *
```

Import packages

```
A = ['hammer', 'wrench', 'screwdriver', 'towel']
v = {'hammer':8, 'wrench':3, 'screwdriver':6, 'towel':11}
w = {'hammer':5, 'wrench':7, 'screwdriver':4, 'towel':3}
W_max = 14
```

Specify/import data

```
model = ConcreteModel()
```

Create model object

```
model.x = Var(A, within=Binary)
```

Define variable x

```
model.value = Objective(
 expr = sum(v[i]*model.x[i] for i in A),
 sense = maximize)
```

Define objective

```
model.weight = Constraint(
 expr = sum(w[i]*model.x[i] for i in A) <= W_max)
```

Define constraint

```
opt = SolverFactory('glpk')
```

Create solver

```
result_obj = opt.solve(model, tee=True)
```

Solve the problem

```
model.pprint()
```

Print results

# Putting it all together: Concrete Knapsack

```
knapsack.py
from pyomo.environ import *
...
result_obj = opt.solve(model, tee=True)
model.pprint()
```

```
> python knapsack.py
```

```
1 Set Declarations
 x_index : Dim=0, Dimen=1, Size=4, Domain=None, Ordered=False, Bounds=None
 ['hammer', 'screwdriver', 'towel', 'wrench']
1 Var Declarations
 x : Size=4, Index=x_index
 Key : Lower : Value : Upper : Fixed : Stale : Domain
 hammer : 0 : 1.0 : 1 : False : False : Binary
 screwdriver : 0 : 1.0 : 1 : False : False : Binary
 towel : 0 : 1.0 : 1 : False : False : Binary
 wrench : 0 : 0.0 : 1 : False : False : Binary
1 Objective Declarations
 value : Size=1, Index=None, Active=True
 Key : Active : Sense : Expression
 None : True : maximize : 8*x[hammer] + 3*x[wrench] + 6*x[screwdriver] + 11*x[towel]
1 Constraint Declarations
 weight : Size=1, Index=None, Active=True
 Key : Lower : Body : Upper : Active
 None : -Inf : 5*x[hammer] + 7*x[wrench] + 4*x[screwdriver] + 3*x[towel] : 14.0 : True
4 Declarations: x_index x value weight
```

# Putting it all together: Concrete Knapsack

```
knapsack.py
from pyomo.environ import *
...
result_obj = opt.solve(model, tee=True)
model.display()
```

```
> python knapsack.py
```

## Variables:

x : Size=4, Index=x\_index

| Key         | : Lower | : Value | : Upper | : Fixed | : Stale | : Domain |
|-------------|---------|---------|---------|---------|---------|----------|
| hammer      | : 0     | : 1.0   | : 1     | : False | : False | : Binary |
| screwdriver | : 0     | : 1.0   | : 1     | : False | : False | : Binary |
| towel       | : 0     | : 1.0   | : 1     | : False | : False | : Binary |
| wrench      | : 0     | : 0.0   | : 1     | : False | : False | : Binary |

## Objectives:

value : Size=1, Index=None, Active=True

| Key  | : Active | : Value |
|------|----------|---------|
| None | : True   | : 25.0  |

## Constraints:

weight : Size=1

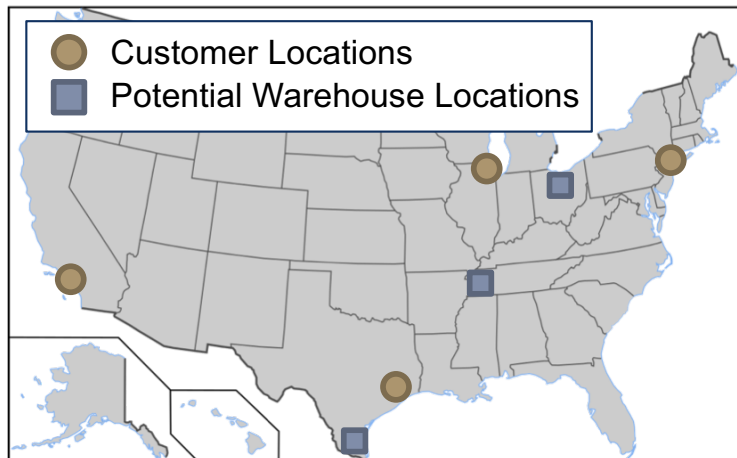
| Key  | : Lower | : Body | : Upper |
|------|---------|--------|---------|
| None | : None  | : 12.0 | : 14.0  |

# Pyomo Fundamentals: Exercises #1

- 1.1 Knapsack example:** Solve the knapsack problem shown in the tutorial using your IDE (e.g., Spyder) or the command line: `> python knapsack.py`. Which items are acquired in the optimal solution? What is the value of the selected items?
- 1.2 Knapsack with improved printing:** The `knapsack.py` example shown in the tutorial uses `model.pprint()` to see the value of the solution variables. Starting with the code in `knapsack_print_incomplete.py`, complete the missing lines to produce formatted output. Note that the Pyomo `value` function should be used to get the floating point value of Pyomo modeling components (e.g., `print(value(model.x[i]))`). Also print the value of the items selected (the objective), and the total weight. (A solution can be found in `knapsack_print_soln.py`).
- 1.3 Changing data:** If we were to increase the value of the wrench, at what point would it become selected as part of the optimal solution? (A solution can be found in `knapsack_wrench_soln.py`.)
- 1.4 Loading data from Excel:** In the knapsack example shown in the tutorial slides, the data is hardcoded at the top of the file. Instead of hardcoding the data, use Python to load the data from a different source. You can start from the file `knapsack_pandas_excel_incomplete.py`. (A solution that uses pandas to load the data from Excel is shown in `knapsack_pandas_excel_soln.py`.)
- 1.5 NLP vs MIP:** Solve the knapsack problem with IPOPT instead of `glpk`. (Hint: switch `glpk` to `ipopt` in the call `SolverFactory`. Print the solution values for `model.x`. What happened? Why?)



# More Complex Example: Warehouse Location



- Determine the set of  $P$  warehouses chosen from the set  $W$  of candidates to minimize the cost of serving all customers  $C$
- Here  $d_{w,c}$  is the cost of serving customer  $c$  from warehouse at location  $w$ .

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s. t. \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C$$

$$x_{w,c} \leq y_w \quad \forall w \in W, c \in C$$

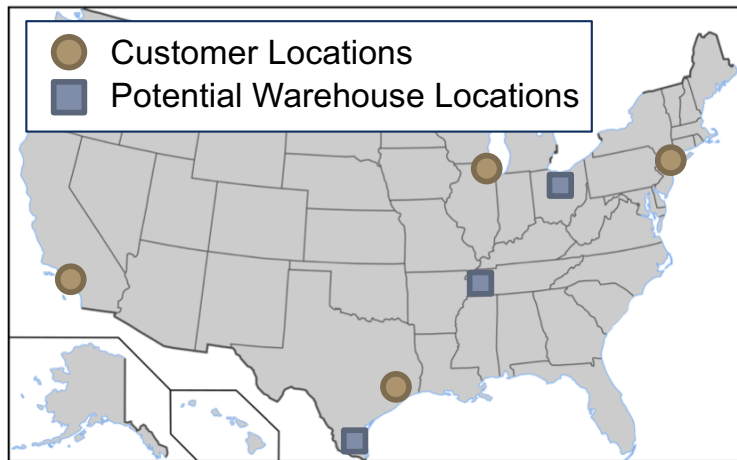
$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|}$$

$y_w$ : discrete variable  
(location  $w$  selected or not)

$x_{w,c}$ : fraction of demand of  
customer  $c$  served by  
warehouse  $w$

# More Complex Example: Warehouse Location



- Determine the set of  $P$  warehouses chosen from the set  $W$  of candidates to minimize the cost of serving all customers  $C$
- Here  $d_{w,c}$  is the cost of serving customer  $c$  from warehouse at location  $w$ .

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c} \quad (\text{minimize total cost})$$

$$s. t. \quad \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C \quad (\text{guarantee all customers served})$$

$$x_{w,c} \leq y_w \quad \forall w \in W, c \in C \quad (\text{customer } c \text{ can only be served from warehouse } w \text{ if warehouse } w \text{ is selected})$$

$$\sum_{w \in W} y_w = P \quad (\text{select } P \text{ warehouses})$$

$$0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|}$$

# Key difference?

## Knapsack

$$\begin{aligned} \max_x \quad & \sum_{i \in A} v_i x_i \\ \text{s.t.} \quad & \sum_{i \in A} w_i x_i \leq W_{\max} \\ & x \in \{0,1\}^{|A|} \end{aligned}$$

## Warehouse Location

$$\begin{aligned} \min \quad & \sum_{w \in W, c \in C} d_{w,c} x_{w,c} \\ \text{s.t.} \quad & \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C \\ & x_{w,c} \leq y_w \quad \forall w \in W, c \in C \\ & \sum_{w \in W} y_w = P \\ & 0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|} \end{aligned}$$

# Key difference: Scalar vs Indexed Constraint

## Knapsack

$$\begin{aligned} \max_x \quad & \sum_{i \in A} v_i x_i \\ \text{s.t.} \quad & \sum_{i \in A} w_i x_i \leq W_{\max} \\ & x \in \{0,1\}^{|A|} \end{aligned}$$

$$w_{\text{hammer}} x_{\text{hammer}} + \cdots + w_{\text{wrench}} x_{\text{wrench}} \leq W_{\max}$$

## Warehouse Location

$$\begin{aligned} \min \quad & \sum_{w \in W, c \in C} d_{w,c} x_{w,c} \\ \text{s.t.} \quad & \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C \\ & x_{w,c} \leq y_w \quad \forall w \in W, c \in C \\ & \sum_{w \in W} y_w = P \\ & 0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|} \end{aligned}$$

# Key difference: Scalar vs Indexed Constraint

## Knapsack

$$\begin{aligned} \max_x \quad & \sum_{i \in A} v_i x_i \\ \text{s.t.} \quad & \sum_{i \in A} w_i x_i \leq W_{\max} \\ & x \in \{0,1\}^{|A|} \end{aligned}$$

$$w_{\text{hammer}} x_{\text{hammer}} + \dots + w_{\text{wrench}} x_{\text{wrench}} \leq W_{\max}$$

$$x_{\text{Har},\text{NYC}} + x_{\text{Mem},\text{NYC}} + x_{\text{Ash},\text{NYC}} = 1$$

$$x_{\text{Har},\text{LA}} + x_{\text{Mem},\text{LA}} + x_{\text{Ash},\text{LA}} = 1$$

$$x_{\text{Har},\text{Chi}} + x_{\text{Mem},\text{Chi}} + x_{\text{Ash},\text{Chi}} = 1$$

$$x_{\text{Har},\text{Hou}} + x_{\text{Mem},\text{Hou}} + x_{\text{Ash},\text{Hou}} = 1$$

## Warehouse Location

$$\begin{aligned} \min \quad & \sum_{w \in W, c \in C} d_{w,c} x_{w,c} \\ \text{s.t.} \quad & \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C \\ & x_{w,c} \leq y_w \quad \forall w \in W, c \in C \\ & \sum_{w \in W} y_w = P \\ & 0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|} \end{aligned}$$

| W \ C     | NYC  | LA   | Chicago | Houston |
|-----------|------|------|---------|---------|
| Harlingen | 1956 | 1606 | 1410    | 330     |
| Memphis   | 1096 | 1792 | 531     | 567     |
| Ashland   | 485  | 2322 | 324     | 1236    |

# Key difference: Scalar vs Indexed Constraint

## Knapsack

$$\begin{aligned} \max_x \quad & \sum_{i \in A} v_i x_i \\ \text{s.t.} \quad & \sum_{i \in A} w_i x_i \leq W_{\max} \\ & x \in \{0,1\}^{|A|} \end{aligned}$$

## Warehouse Location

$$\begin{aligned} \min \quad & \sum_{w \in W, c \in C} d_{w,c} x_{w,c} \\ \text{s.t.} \quad & \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C \\ & x_{w,c} \leq y_w \quad \forall w \in W, c \in C \\ & \sum_{w \in W} y_w = P \\ & 0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|} \end{aligned}$$

- Knapsack problem: All constraints are singular (i.e., scalar)
- Warehouse location problem:
  - Multiple constraints defined over indices
- Pyomo used the concept of *construction rules* to specify indexed constraints

# Indexed Constraints: Construction Rules Sandia National Laboratories

$$\sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C$$

```
W = ['Harlingen', 'Memphis', 'Ashland']
```

```
C = ['NYC', 'LA', 'Chicago', 'Houston']
```

```
model.x = Var(W, C, bounds=(0,1))
```

```
model.y = Var(W, within=Binary)
```

Particular index is passed as argument to the rule

```
def one_per_cust_rule(m, c):
```

```
 return sum(m.x[w,c] for w in W) == 1
```

```
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)
```

For indexed constraints, you provide a “rule” (function) that returns an expression (or tuple) for each index.

Rule is called once for every entry in C!

# Construction Rules: Multiple indices

$$x_{w,c} \leq y_w \quad \forall w \in W, c \in C$$

```
W = ['Harlingen', 'Memphis', 'Ashland']
```

```
C = ['NYC', 'LA', 'Chicago', 'Houston']
```

```
model.x = Var(W, C, bounds=(0,1))
```

```
model.y = Var(W, within=Binary)
```

Each dimension of the indices is  
a separate argument to the rule

```
def warehouse_active_rule(m, w, c):
```

```
 return m.x[w,c] <= m.y[w]
```

```
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)
```

Rule is called once for every entry  $w$  in  $W$   
crossed with every entry  $c$  in  $C$ !



# Construction Rules: Complex logic

$$x_i = \begin{cases} \cos(y_i), & i \text{ even} \\ \sin(y_i), & i \text{ odd} \end{cases} \quad \forall \{i \in N: i \neq 0\}$$

```
def complex_rule(model, i):
 if i == 0:
 return Constraint.Skip
 elif i % 2 == 0:
 return model.x[i] == cos(model.y[i])
 return model.x[i] == sin(model.y[i])
model.complex_constraint = Constraint(N, M, rule=complex_rule)
```

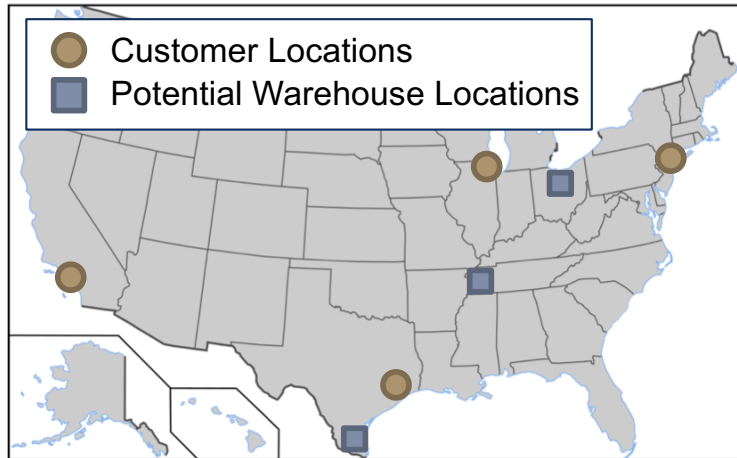
Constraint rules can contain complex logic on the indices and other parameters, but NOT on the value of model variables.

# More on *Construction Rules*

---

- Components are constructed in declaration order
  - The instructions for *how* to construct the object are provided through a function, or *rule*
  - Pyomo calls the rule for each component index
  - *Rules* can be provided to virtually all Pyomo components
- Naming conventions
  - the component name prepended with “\_” ( $c4 \rightarrow \_c4$ )
  - the component name with “\_rule” appended ( $c4 \rightarrow c4\_rule$ )
  - each rule is called “rule” (Python implicitly overrides each declaration)
- Abstract models (discussed later) construct the model in two passes:
  - Python parses the model declaration
    - creating “empty” Pyomo components in the model
  - Pyomo loads and parses external data

# Putting it all together: Warehouse Location



- Determine the set of  $P$  warehouses chosen from the set  $W$  of candidates to minimize the cost of serving all customers  $C$
- Here  $d_{w,c}$  is the cost of serving customer  $c$  from warehouse at location  $w$ .

$$\min \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

$$s. t. \sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C$$

$$x_{w,c} \leq y_w \quad \forall w \in W, c \in C$$

$$\sum_{w \in W} y_w = P$$

$$0 \leq x \leq 1 \quad y \in \{0,1\}^{|W|}$$

| W         | C | NYC  | LA   | Chicago | Houston |
|-----------|---|------|------|---------|---------|
| Harlingen |   | 1956 | 1606 | 1410    | 330     |
| Memphis   |   | 1096 | 1792 | 531     | 567     |
| Ashland   |   | 485  | 2322 | 324     | 1236    |

$$P = 2$$

# Putting It All Together: Warehouse Location

```
warehouse location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

# Putting It All Together: Warehouse Location

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

# Putting It All Together: Warehouse Location

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

# Putting It All Together: Warehouse Location

$$x_{w,c} \quad \forall w \in W, c \in C$$

$$y_w \quad \forall w \in W$$

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *
```

```
model = ConcreteModel(name="(WL)")
```

```
W = ['Harlingen', 'Memphis', 'Ashland']
```

```
C = ['NYC', 'LA', 'Chicago', 'Houston']
```

```
d = {('Harlingen', 'NYC'): 1956, \
```

```
 . . .
```

```
 ('Ashland', 'Houston'): 1236 }
```

```
P = 2
```

```
model.x = Var(W, C, bounds=(0,1))
```

```
model.y = Var(W, within=Binary)
```

```
def obj_rule(m):
```

```
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
```

```
model.obj = Objective(rule=obj_rule)
```

```
def one_per_cust_rule(m, c):
```

```
 return sum(m.x[w,c] for w in W) == 1
```

```
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)
```

```
def warehouse_active_rule(m, w, c):
```

```
 return m.x[w,c] <= m.y[w]
```

```
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)
```

```
def num_warehouses_rule(m):
```

```
 return sum(m.y[w] for w in W) <= P
```

```
model.num_warehouses = Constraint(rule=num_warehouses_rule)
```

```
SolverFactory('glpk').solve(model)
```

```
model.pprint()
```

# Putting It All Together: Warehouse Location

$$\min_{x,y} \sum_{w \in W, c \in C} d_{w,c} x_{w,c}$$

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *
```

```
model = ConcreteModel(name="(WL)")
```

```
W = ['Harlingen', 'Memphis', 'Ashland']
```

```
C = ['NYC', 'LA', 'Chicago', 'Houston']
```

```
d = {('Harlingen', 'NYC'): 1956, \
```

```
 . . .
```

```
 ('Ashland', 'Houston'): 1236 }
```

```
P = 2
```

```
model.x = Var(W, C, bounds=(0,1))
```

```
model.y = Var(W, within=Binary)
```

```
def obj_rule(m):
```

```
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
```

```
model.obj = Objective(rule=obj_rule)
```

```
def one_per_cust_rule(m, c):
```

```
 return sum(m.x[w,c] for w in W) == 1
```

```
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)
```

```
def warehouse_active_rule(m, w, c):
```

```
 return m.x[w,c] <= m.y[w]
```

```
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)
```

```
def num_warehouses_rule(m):
```

```
 return sum(m.y[w] for w in W) <= P
```

```
model.num_warehouses = Constraint(rule=num_warehouses_rule)
```

```
SolverFactory('glpk').solve(model)
```

```
model.pprint()
```



# Putting It All Together: Warehouse Location

$$\sum_{w \in W} x_{w,c} = 1 \quad \forall c \in C$$

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

# Putting It All Together: Warehouse Location

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

$$x_{w,c} \leq y_w \quad \forall w \in W, c \in C$$

# Putting It All Together: Warehouse Location

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

$$\sum_{w \in W} y_w = P$$

# Putting It All Together: Warehouse Location

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

# Putting It All Together: Warehouse Location

```
warehouse_location.py: Warehouse location determination problem
from pyomo.environ import *

model = ConcreteModel(name="(WL)")

W = ['Harlingen', 'Memphis', 'Ashland']
C = ['NYC', 'LA', 'Chicago', 'Houston']
d = {('Harlingen', 'NYC'): 1956, \
 . . .
 ('Ashland', 'Houston'): 1236 }
P = 2

model.x = Var(W, C, bounds=(0,1))
model.y = Var(W, within=Binary)

def obj_rule(m):
 return sum(d[w,c]*m.x[w,c] for w in W for c in C)
model.obj = Objective(rule=obj_rule)

def one_per_cust_rule(m, c):
 return sum(m.x[w,c] for w in W) == 1
model.one_per_cust = Constraint(C, rule=one_per_cust_rule)

def warehouse_active_rule(m, w, c):
 return m.x[w,c] <= m.y[w]
model.warehouse_active = Constraint(W, C, rule=warehouse_active_rule)

def num_warehouses_rule(m):
 return sum(m.y[w] for w in W) <= P
model.num_warehouses = Constraint(rule=num_warehouses_rule)

SolverFactory('glpk').solve(model)

model.pprint()
```

# Pyomo Fundamentals: Exercises #2

**2.1 Knapsack problem with rules:** Rules are important for defining indexed constraints, however, they can also be used for single (i.e. scalar) constraints. Starting with `knapsack.py`, reimplement the model using rules for the objective and the constraints. (A solution can be found in `knapsack_rules_soln.py`.)

**2.2 Integer formulation of the knapsack problem:** Consider again, the knapsack problem. Assume now that we can acquire multiple items of the same type. In this new formulation,  $x_i$  is now an integer variable instead of a binary variable. One way to formulate this problem is as follows:

$$\begin{aligned} \max_{q,x} \quad & \sum_{i \in A} v_i x_i \\ \text{s.t.} \quad & \sum_{i \in A} w_i x_i \leq W_{\max} \\ & x_i = \sum_{j=0}^N j q_{i,j} & \forall i \in A \\ & 0 \leq x \leq N \\ & q_{i,j} \in \{0, 1\} & \forall i \in A, j \in \{0..N\} \end{aligned}$$

Starting with `knapsack_rules.py`, implement this new formulation and solve. Is the solution surprising? (A solution can be found in `knapsack_integer_soln.py`.)

# Parameters

- Pyomo models can be built using standard python numeric types (e.g., int, float) for all constants/data
  - This is how we have done examples so far.
- Pyomo also supports a parameter component (Param)
  - Keeps data documented on the model
  - Allows for validation of data, default values, and changes in data without the need to rebuild entire model
  - Allows Abstract model definitions (declare model, apply data later)

Provide an (initial) value of 42 for the parameter

- Scalar numeric values

`model.a_parameter = Param( initialize = 42 )`

# Parameters

- Indexed numeric values

```
model.a_param_vec = Param(IDX,
 initialize = data,
 default = 0)
```

Providing “default” allows the initialization data to only specify the “unusual” values

“data” *must* be a dictionary(\*) of index keys to values because all sets are assumed to be *unordered*

(\*) – *actually, it must define \_\_getitem\_\_(), but that only really matters to Python geeks*

- Mutable Parameters

```
model.a_parameter = Param(initialize = 42,
 mutable = True)
```

Indicates to Pyomo that you may want to change this parameter later.



# Generating and Managing Indices: *Sets*

- Any iterable object can be an index, e.g., lists:
  - `IDX_a = [1,2,5]`
  - `DATA = {1: 10, 2: 21, 5:42};`  
`IDX_b = DATA.keys()`
- Sets: objects for managing multidimensional indices
  - `model.IDX = Set( initialize = [1,2,5] )`

Note: capitalization matters:  
**Set** = Pyomo class  
**set** = native Python set

Like indices, Sets can be  
initialized from any iterable

- `model.IDX = Set( [1,2,5] )`

**Note:** This doesn't do what you want.  
This creates a 3-member *indexed set*, where each set is *empty*.

# Sequential Indices: *RangeSet*

- Sets of sequential integers are common

- `model.IDX = Set( initialize=range(5) )`

- `model.IDX = RangeSet( 5 )`

Note: RangeSet is 1-based.  
This gives [ 1, 2, 3, 4, 5 ]

Note: Python range is 0-based.  
This gives [ 0, 1, 2, 3, 4 ]

- You can provide lower and upper bounds to RangeSet

- `model.IDX = RangeSet( 0, 4 )`

This gives [ 0, 1, 2, 3, 4 ]

# Manipulating Sets

- Creating sparse sets

```
model.IDX = Set(initialize=[1,2,5])
def lower_tri_filter(model, i, j):
 return j <= i
model.LTRI = Set(initialize = model.IDX * model.IDX,
 filter = lower_tri_filter)
```

The filter should return *True* if the element is in the set; *False* otherwise.

- Sets support efficient higher-dimensional indices

```
model.IDX = Set(initialize=[1,2,5])
model.IDX2 = model.IDX * model.IDX
```

This creates a *virtual*  
2-D “matrix” Set

Sets also support union (&), intersection (|),  
difference (-), symmetric difference (^)

# Higher-Dimensional Sets and Flattening

- Higher-dimensional sets contain multiple indices per element

```
model.IDX = Set(initialize=[1,2,5])
model.IDX2 = model.IDX * model.IDX
```

```
tuples = list()
for i in model.IDX:
 for j in model.IDX:
 tuples.append((i,j))
model.IDXT = Set(initialize=tuples)
```

\* IDX2 and IDXT represent [ (1,1), (1,2), (1,5), (2,1), (2,2), (2,5), (5,1), (5,2), (5,5) ]

- Higher-dimensional sets and rules

```
def c5_rule(model, i, j, k):
 return model.a[i] + model.a[j] + model.a[k] <= 1
model.c5 = Constraint(model.IDX2, model.IDX, rule=c5_rule)
```

Each dimension of each index is a separate argument to the rule

# Set ordering

- By default, Sets are *unordered* (just like Python sets and dictionaries)

```
model.IDX = Set(initialize=[1,2,5])
```

```
for i in model.IDX:
 print(i)
```

← No specific order guaranteed for this loop  
(e.g., 1,5,2 ... 5,1,2 ...)

```
model.x = Var(model.IDX)
```

```
for k in model.x:
 print(value(model.x[i]))
```

← This is also true for variables indexed by unordered sets.

```
model.IDX0 = Set(initialize=[1,2,5], ordered=True)
```

↑ Use the keyword argument `ordered=True` to guarantee fixed order.

# Other Modeling Components

- Pyomo supports “list”-like indexed components (useful for meta-algorithms and addition of cuts)

```
model.a = Var()
model.b = Var()
model.c = Var()
model.limits = ConstraintList()
```

```
model.limits.add(30*model.a + 15*model.b + 10*model.c <= 100)
model.limits.add(10*model.a + 25*model.b + 5*model.c <= 75)
model.limits.add(6*model.a + 11*model.b + 3*model.c <= 30)
```

↑  
“add” adds a single new constraint to the list.  
The constraints need not be related.

# Pyomo Fundamentals: Exercises #3

**3.1 Changing Parameter values:** In the tutorial slides, we saw that a parameter could be specified to be `mutable`. This tells Pyomo that the value of the parameter may change in the future, and allows the user to change the parameter value and resolve the problem without the need to rebuild the entire model each time. We will use this functionality to find a better solution to an earlier exercise. Considering again the knapsack problem, we would like to find when the wrench becomes valuable enough to be a part of the optimal solution. Create a Pyomo `Parameter` for the value of the items, make it mutable, and then write a loop that prints the solution for different wrench values. Start with the file `knapsack_mutable_parameter_incomplete.py`. (A solution for this problem can be found in `knapsack_mutable_parameter_soln.py`.)

**3.2 Integer cuts:** Often, it can be important to find not only the “best” solution, but a number of solutions that are equally optimal, or close to optimal. For discrete optimization problems, this can be done using something known as an integer cut. Consider again the knapsack problem where the choice of which items to select is a discrete variable  $x_i \forall i \in A$ . Let  $x_i^*$  be a particular set of  $x$  values we want to remove from the feasible solution space. We define an integer cut using two sets. The first set  $S_0$  contains the indices for those variables whose current solution is 0, and the second set  $S_1$  consists of indices for those variables whose current solution is 1. Given these two sets, an integer cut constraint that would prevent such a solution from appearing again is defined by,

$$\sum_{i \in S_0} x[i] + \sum_{i \in S_1} (1 - x[i]) \geq 1.$$

Starting with `knapsack_rules.py`, write a loop that solves the problem 5 times, adding an integer cut to remove the previous solution, and printing the value of the objective function and the solution at each iteration of the loop. (A solution for this problem can be found in `knapsack_integer_cut_soln.py`)

# Pyomo Fundamentals: Exercises #3

**3.3 Putting it all together with the lot sizing example:** We will now write a complete model from scratch using a well-known multi-period optimization problem for optimal lot-sizing adapted from Hagen et al. (2001) shown below.

$$\min \sum_{t \in T} c_t y_t + h_t^+ I_t^+ + h_t^- I_t^- \quad (1)$$

$$\text{s.t. } I_t = I_{t-1} + X_t - d_t \quad \forall t \in T \quad (2)$$

$$I_t = I_t^+ - I_t^- \quad \forall t \in T \quad (3)$$

$$X_t \leq P y_t \quad \forall t \in T \quad (4)$$

$$X_t, I_t^+, I_t^- \geq 0 \quad \forall t \in T \quad (5)$$

$$y_t \in \{0, 1\} \quad \forall t \in T \quad (6)$$

Our goal is to find the optimal production  $X_t$  given known demands  $d_t$ , fixed cost  $c_t$  associated with active production in a particular time period, an inventory holding cost  $h_t^+$  and a shortage cost  $h_t^-$  (cost of keeping a backlog) of orders. The variable  $y_t$  (binary) determines if we produce in time  $t$  or not, and  $I_t^+$  represents inventory that we are storing across time period  $t$ , while  $I_t^-$  represents the magnitude of the backlog. Note that equation (4) is a constraint that only allows production in time period  $t$  if the indicator variable  $y_t=1$ .

Write a Pyomo model for this problem and solve it using `glpk` using the data provided below. You can start with the file `lot_sizing_incomplete.py`. (A solution is provided in `lot_sizing_soln.py`.)

| Parameter | Description                             | Value                 |
|-----------|-----------------------------------------|-----------------------|
| $c$       | fixed cost of production                | 4.6                   |
| $I_0^+$   | initial value of positive inventory     | 5.0                   |
| $I_0^-$   | initial value of backlogged orders      | 0.0                   |
| $h^+$     | cost (per unit) of holding inventory    | 0.7                   |
| $h^-$     | shortage cost (per unit)                | 1.2                   |
| $P$       | maximum production amount (big-M value) | 5                     |
| $d$       | demand                                  | [5, 7, 6.2, 3.1, 1.7] |



# Other Topics

---

# Solving models: *the pyomo command*

- `pyomo` (`pyomo.exe` on Windows):
  - Constructs model and passes it to an (external) solver

```
pyomo solve <model_file> [<data_file> ...] [options]
```

- Installed to:
  - `[PYTHONHOME]\Scripts` [Windows; `C:\Python27\Scripts`]
  - `[PYTHONHOME]/bin` [Linux; `/usr/bin`]
- Key options (*many* others; see `--help`)

|                                                 |                                                                                            |
|-------------------------------------------------|--------------------------------------------------------------------------------------------|
| <code>--help</code>                             | Get list of all options                                                                    |
| <code>--help-solvers</code>                     | Get the list of all recognized solvers                                                     |
| <code>--solver=&lt;solver_name&gt;</code>       | Set the solver that Pyomo will invoke                                                      |
| <code>--solver-options="key=value[ ...]"</code> | Specify options to pass to the solver as a space-separated list of keyword-value pairs     |
| <code>--stream-solver</code>                    | Display the solver output during the solve                                                 |
| <code>--summary</code>                          | Display a summary of the optimization result                                               |
| <code>--report-timing</code>                    | Report additional timing information, including construction time for each model component |

# *Abstract Modeling*

---

# Concrete vs. Abstract Models

---

- Concrete Models: data first, then model
  - 1-pass construction
  - All data must be present before Python starts processing the model
  - Pyomo will construct each component in order at the time it is declared
  - Straightforward logical process; easy to script.
  - Familiar to modelers with experience with GAMS
  
- Abstract Models: model first, then data
  - 2-pass construction
  - Pyomo stores the basic model declarations, but does not construct the actual objects
    - Details on how to construct the component hidden in functions, or *rules*
    - e.g., it will declare an indexed variable “x”, but will not expand the indices or populate any of the individual variable values.
  - At “creation time”, data is applied to the abstract declaration to create a concrete instance (components are still constructed in declaration order)
  - Encourages generic modeling and model reuse
    - e.g., model can be used for arbitrary-sized inputs
  - Familiar to modelers with experience with AMPL

# Data Sources

---

- Data can be imported from “.dat” file

- Format similar to AMPL style
- Explicit data from “param” declarations
- External data through “load” declarations:

- Excel

- ```
load ABCD.xls range=ABCD : Z=[A, B, C] Y=D ;
```

- Databases

- ```
load “DBQ=diet.mdb” using=pyodbc query=“SELECT FOOD, cost,
f_min, f_max from Food” : [FOOD] cost f_min f_max ;
```

- External data overrides “initialize=” declarations

# Abstract p-Median (pmedian.py, 1)

---

```
from pyomo.environ import *

model = AbstractModel()

model.N = Param(within=PositiveIntegers)
model.P = Param(within=RangeSet(model.N))
model.M = Param(within=PositiveIntegers)

model.Locations = RangeSet(model.N)
model.Customers = RangeSet(model.M)

model.d = Param(model.Locations, model.Customers)

model.x = Var(model.Locations, model.Customers, bounds=(0.0, 1.0))
model.y = Var(model.Locations, within=Binary)
```

# Abstract p-Median (pmedian.py, 2)

```
def obj_rule(model):
 return sum(model.d[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers)
model.obj = Objective(rule=obj_rule)

def single_x_rule(model, m):
 return sum(model.x[n,m] for n in model.Locations) == 1.0
model.single_x = Constraint(model.Customers, rule=single_x_rule)

def bound_y_rule(model, n,m):
 return model.x[n,m] - model.y[n] <= 0.0
model.bound_y = Constraint(model.Locations, model.Customers,
 rule=bound_y_rule)

def num_facilities_rule(model):
 return sum(model.y[n] for n in model.Locations) == model.P
model.num_facilities = Constraint(rule=num_facilities_rule)
```

# Abstract p-Median (pmedian.dat)

---

```
param N := 3;
```

```
param M := 4;
```

```
param P := 2;
```

```
param d: 1 2 3 4 :=
 1 1.7 7.2 9.0 8.3
 2 2.9 6.3 9.8 0.7
 3 4.5 4.8 4.2 9.3 ;
```



# In Class Exercise: Abstract Knapsack

$$\begin{aligned} \max \quad & \sum_{i=1}^N v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^N w_i x_i \leq W_{\max} \\ & x_i \in \{0,1\} \end{aligned}$$

| Item        | Weight | Value |
|-------------|--------|-------|
| hammer      | 5      | 8     |
| wrench      | 7      | 3     |
| screwdriver | 4      | 6     |
| towel       | 3      | 11    |

Max weight: 14

Syntax reminders:

`AbstractModel()`

`Set( [index, ...], [initialize=list/function] )`

`Param( [index, ...], [within=domain], [initialize=dict/function] )`

`Var( [index, ...], [within=domain], [bounds=(lower,upper)] )`

`Constraint( [index, ...], [expr=expression/rule=function] )`

`Objective( sense={maximize/minimize},  
          expr=expression/rule=function )`

# Abstract Knapsack: *Solution*

```
from pyomo.environ import *

model = AbstractModel()
model.ITEMS = Set()
model.v = Param(model.ITEMS, within=PositiveReals)
model.w = Param(model.ITEMS, within=PositiveReals)
model.W_max = Param(within=PositiveReals)
model.x = Var(model.ITEMS, within=Binary)

def value_rule(model):
 return sum(model.v[i]*model.x[i] for i in model.ITEMS)
model.value = Objective(rule=value_rule, sense=maximize)

def weight_rule(model):
 return sum(model.w[i]*model.x[i] for i in model.ITEMS) \
 <= model.W_max
model.weight = Constraint(rule=weight_rule)
```

# Abstract Knapsack: *Solution Data*

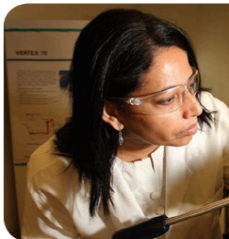
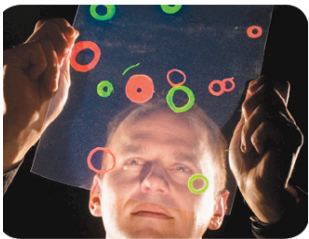
---

```
set ITEMS := hammer wrench screwdriver towel ;
```

```
param: v w :=
```

|             |    |    |
|-------------|----|----|
| hammer      | 8  | 5  |
| wrench      | 3  | 7  |
| screwdriver | 6  | 4  |
| towel       | 11 | 3; |

```
param W_max := 14;
```



*Exceptional  
service  
in the  
national  
interest*

## 4. Pyomo Idioms



U.S. DEPARTMENT OF  
**ENERGY**



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Pyomo and Python: Idioms and Efficiency

---

- Being embedded in a high-level (and interpreted) programming language can present challenges
  - Inability to constrain syntax => users have many guns
  - Some of approaches may be *very* slow
- Some of the blame can be placed on Python
  - But a lot can be blamed on Pyomo

# What are *reasonable* performance expectations?

---

- Python is a *byte-compiled scripting language*
  - and Pyomo is pure Python
  - ...so expectations were not high
  - *...and raw speed has never been a goal!*
- Early experiences bore this out... in November, 2010:
  - *p*-median facility location
    - AMPL model construction time: ~4 seconds
    - Pyomo model construction time: >2000 seconds
  - Logistics disruption modeling
    - GAMS solution time: ~20 seconds
    - Pyomo solution time: >200 seconds
- ...but the gap is closing... in Coopr 3.2:
  - *p*-median facility location: ~45 seconds
  - Logistics disruption modeling: ~25 seconds

# Managing edge cases (special sets vs rule logic)

- Linking time; consider:

$$T \in [0..T_{\max}]$$

$$x_t = cx_{t-1} \quad \{t \mid t \in T, t \neq 0\}$$

# Managing edge cases (special sets vs rule logic)

- Linking time; consider:

$$T \in [0..T_{\max}]$$
$$x_t = cx_{t-1} \quad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)
```

```
model.Tnot0 = model.T - [0]
```

```
def rule1(model, t):
 return model.x[t] == model.c * model.x[t-1]
model.link = Constraint(model.Tnot0, rule=rule1)
```



# Managing edge cases (special sets vs rule logic)

- Linking time; consider:

$$T \in [0..T_{\max}]$$
$$x_t = cx_{t-1} \quad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)
```

```
def rule2(model, t):
 if t == 0:
 return Constraint.Skip
 return model.x[t] == model.c * model.x[t-1]
model.link = Constraint(model.T, rule=rule2)
```

# Managing edge cases (special sets vs rule logic)

- Linking time; consider:

$$T \in [0..T_{\max}]$$
$$x_t = cx_{t-1} \quad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)
```

```
model.Tnot0 = model.T - [0]
```

```
def rule1(model, t):
 return model.x[t] == model.c * model.x[t-1]
model.link = Constraint(model.Tnot0, rule=rule1)
```

```
def rule2(model, t):
 if t == 0:
 return Constraint.Skip
 return model.x[t] == model.c * model.x[t-1]
model.link = Constraint(model.T, rule=rule2)
```

# Managing edge cases (special sets vs rule logic)

- Linking time; consider:

$$T \in [0..T_{\max}]$$
$$x_t = cx_{t-1} \quad \{t \mid t \in T, t \neq 0\}$$

```
model.T = RangeSet(0, model.Tmax)
```

```
model.Tnot0 = model.T - [0]
```

```
def rule1(model, t):
 return model.x[t] == model.c * model.x[t-1]
model.link = Constraint(model.Tnot0, rule=rule1)
```

```
def rule2(model, t):
 if t == 0:
 return Constraint.Skip
 return model.x[t] == model.c * model.x[t-1]
model.link = Constraint(model.T, rule=rule2)
```

[ Tmax = 1e5 ]  
rule1: 4.35 sec  
rule2: 4.10 sec

# Managing performance (how not to shoot yourself)

---

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

# Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

```
def rule1(model):
 ans = 0
 for n in model.Locations:
 for m in model.Customers:
 ans = ans + model.d[n,m]*model.x[n,m]
 return ans
```

```
model.obj = Objective(rule=ruleN)
```

# Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

```
def rule2(model):
 ans = 0
 for n in model.Locations:
 for m in model.Customers:
 ans += model.d[n,m]*model.x[n,m]
 return ans
```

```
model.obj = objective(rule=ruleN)
```

# Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

```
def rule3(model):
 return sum([model.d[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers])
```

```
model.obj = Objective(rule=ruleN)
```

# Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

```
def rule4(model):
 return sum(model.d[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers)

model.obj = Objective(rule=ruleN)
```



# Managing performance (how not to shoot yourself)

- Expression generation; consider:

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

```
def rule1(model):
 ans = 0
 for n in model.Locations:
 for m in model.Customers:
 ans = ans + model.d[n,m]*model.x[n,m]
 return ans

def rule2(model):
 ans = 0
 for n in model.Locations:
 for m in model.Customers:
 ans += model.d[n,m]*model.x[n,m]
 return ans

def rule3(model):
 return sum([model.d[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers])

def rule4(model):
 return sum(model.d[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers)

model.obj = Objective(rule=ruleN)
```

[ n = m = 1..640 ]

rule1: >>10000 sec  
rule2: 5.0 sec  
rule3: 7.4 sec  
rule4: 5.0 sec

# Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall n \in N$$

```
model.a = Param(model.N, model.M, default=0, mutable=True)
```

```
def rule1(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M) <= model.b[n])
```

For  $n = 1..10$ ,  $m = 1..1e5$ , 4% nonzero,  
1.9 seconds to generate the constraint  
1e5 terms in the constraint (dense!!)

```
model.C = Constraint(model.N, rule=ruleN)
```

# Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall n \in N$$

```
model.a = Param(model.N, model.M, default=0, mutable=True)
```

```
def rule2(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if model.a[n,m] != 0) <= model.b[n]
```

For  $n = 1..10$ ,  $m = 1..1e5$ , 4% nonzero,  
0.1 seconds *slower*, and still dense!

```
model.C = Constraint(model.N, rule=ruleN)
```

# Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall n \in N$$

```
model.a = Param(model.N, model.M, default=0, mutable=True)
```

```
def rule3(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if value(model.a[n,m]) != 0) <= model.b[n]
```

```
def rule4(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if model.a[n,m].value != 0) <= model.b[n]
```

```
model.C = Constraint(model.N, rule=ruleN)
```

# Managing performance (how not to shoot yourself)

- Sparse data; consider:

$$\sum_{m \in M} a_{n,m} x_m \leq b_n \quad \forall n \in N$$

```
model.a = Param(model.N, model.M, default=0, mutable=True)
```

```
def rule1(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M) <= model.b[n])
```

```
def rule2(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if model.a[n,m] != 0) <= model.b[n]
```

```
def rule3(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if value(model.a[n,m]) != 0) <= model.b[n]
```

```
def rule4(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if model.a[n,m].value != 0) <= model.b[n]
```

```
model.C = Constraint(model.N, rule=ruleN)
```

[ n = 1..10, m = 1..1e5,  
 4% fill ]

rule1: 1.9 sec

rule2: 2.0 sec

rule3: 0.9 sec

rule4: 0.8 sec

# Managing performance (how not to shoot yourself)

## ■ Mutable vs Immutable Params

- Given a promise parameter value *will never change*, we can optimize
- Immutable Params are now the default!

```
model.a = Param(model.N, model.M, default=0, mutable=False)
```

```
def rule1(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M) <= model.b[n])
```

```
def rule2(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if model.a[n,m] != 0) <= model.b[n]
```

```
def rule3(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if value(model.a[n,m]) != 0) <= model.b[n]
```

```
def rule4(model,n):
 return sum(model.a[n,m] * model.x[m] for m in model.M
 if model.a[n,m].value != 0) <= model.b[n]
```

```
model.C = Constraint(model.N, rule=ruleN)
```

[ n = 1..10, m = 1..1e5,  
4% fill ]

rule1: 1.2 sec

rule2: 0.5 sec

rule3: 0.5 sec

rule4: *Exception!*

# Managing performance (how not to shoot yourself)

---

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall t \in [dStart_d, dEnd_d], p \in Products, d \in Disruptions$$

# Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall t \in [dStart_d, dEnd_d], p \in Products, d \in Disruptions$$

```
def rule1(model,t,d,p):
 if t < model.dStart[d] or t > model.dEnd[d]:
 return Constraint.Skip
 return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]
model.C1 = Constraint(model.TIME, model.DISRUPTIONS, model.PRODUCTS, rule=rule1)
```



# Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall t \in [dStart_d, dEnd_d], p \in Products, d \in Disruptions$$

```
def rule2(model,t,d,p):
 return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

def _filter2(model,t,d,p):
 return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set(within=model.TIME * model.DISRUPTIONS * model.PRODUCTS,
 filter=_filter2)
model.C2 = Constraint(model.ACTIVE_DISRUPTIONS, rule=rule2)
```

# Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall t \in [dStart_d, dEnd_d], p \in Products, d \in Disruptions$$

```
def rule2(model,t,d,p):
 return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]
```

```
def _filter3(model,t,d):
 return t >= model.dStart[d] and t <= model.dEnd[d]
model.ACTIVE_DISRUPTIONS = Set(within=model.TIME * model.DISRUPTIONS, filter=_filter3)
model.C3 = Constraint(model.ACTIVE_DISRUPTIONS, model.PRODUCTS, rule=rule2)
```

# Managing performance (how not to shoot yourself)

- Sparse constraints; consider:

$$storage_{t,p,d} = storage_{t-1,p,d} + production_{t,p,d}$$

$$\forall t \in [dStart_d, dEnd_d], p \in Products, d \in Disruptions$$

[ t = d = 1..250,  
p = 1..10,  
t\*d = 2% fill ]

C1: 2.2 sec

C2: 2e-4 sec

C3: 2e-4 sec

```
def rule1(model,t,d,p):
 if t < model.dStart[d] or t > model.dEnd[d]:
 return Constraint.Skip
 return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

model.C1 = Constraint(model.TIME, model.DISRUPTIONS, model.PRODUCTS, rule1)

def rule2(model,t,d,p):
 return model.storage[t,p,d] == model.storage[t-1,p,d] + model.production[t,p,d]

def _filter2(model,t,d,p):
 return t >= model.dStart[d] and t <= model.dEnd[d]

model.ACTIVE_DISRUPTIONS = Set(within=model.TIME * model.DISRUPTIONS * model.PRODUCTS,
 filter=_filter2)

model.C2 = Constraint(model.ACTIVE_DISRUPTIONS, rule=rule2)

def _filter3(model,t,d):
 return t >= model.dStart[d] and t <= model.dEnd[d]

model.ACTIVE_DISRUPTIONS = Set(within=model.TIME * model.DISRUPTIONS, filter=_filter3)

model.C3 = Constraint(model.ACTIVE_DISRUPTIONS, model.PRODUCTS, rule=rule2)
```

# Managing performance (how not to shoot yourself)

- Using Pyomo convenience functions

$$\min \sum_n \sum_m d_{n,m} x_{n,m}, \quad n \in \text{Locations}, m \in \text{Customers}$$

```
model.d1 = Param(model.Locations, model.Customers, mutable=True)
```

```
def rule1(model):
 return sum(model.d1[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers)
```

```
def rule2(model):
 return summation(model.d1, model.x)
```

```
model.d2 = Param(model.Locations, model.Customers, mutable=False)
```

```
def rule3(model):
 return sum(model.d2[n,m]*model.x[n,m]
 for n in model.Locations for m in model.Customers)
```

```
def rule4(model):
 return sumation(model.d2, model.x)
```

[ n = m = 1..640 ]

rule1: 5.0 sec

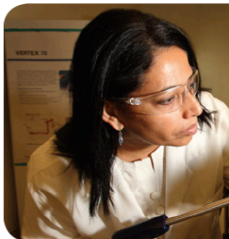
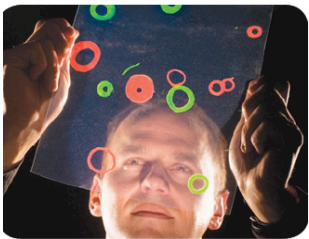
rule2: 7.9 sec

rule3: 3.5 sec

rule4: 5.6 sec

# The Performance “Elephant”: Memory

- Known issue ...
  - Python uses a fairly heavy-weight object model
  - “Significant” recent improvements in low-level core components
    - Coopr 3.3 uses <50% of the memory of Coopr 3.0
  - But...
    - 640 x 640  $p$ -median problem still consumes ~1 GB.
  
- Focus of efforts, with several more enhancements on the horizon.



*Exceptional  
service  
in the  
national  
interest*

## 5. Nonlinear Problems



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Nonlinear problems are easy...

---

# Nonlinear problems are easy...

---

... to write in Pyomo (correct formulation and solution is another story)



# Nonlinear: Supported expressions

| Operation                            | Operator         | Example                                    |
|--------------------------------------|------------------|--------------------------------------------|
| multiplication                       | *                | <code>expr = model.x * model.y</code>      |
| division                             | /                | <code>expr = model.x / model.y</code>      |
| exponentiation                       | **               | <code>expr = (model.x+2.0)**model.y</code> |
| in-place multiplication <sup>1</sup> | <code>*=</code>  | <code>expr *= model.x</code>               |
| in-place division <sup>2</sup>       | <code>/=</code>  | <code>expr /= model.x</code>               |
| in-place exponentiation <sup>3</sup> | <code>**=</code> | <code>expr **= model.x</code>              |

```

model = ConcreteModel()
model.r = Var()
model.h = Var()

def surf_area_obj_rule(m):
 return 2 * math.pi * m.r * (m.r + m.h)
model.surf_area_obj = Objective(rule=surf_area_obj_rule)

def vol_con_rule(m):
 return pi * m.h * m.r**2 == 355
model.vol_con = Constraint(rule=vol_con_rule)

```

# Nonlinear: Supported expressions

| Operation             | Function | Example               |
|-----------------------|----------|-----------------------|
| arccosine             | acos     | expr = acos(model.x)  |
| hyperbolic arccosine  | acosh    | expr = acosh(model.x) |
| arcsine               | asin     | expr = asin(model.x)  |
| hyperbolic arcsine    | asinh    | expr = asinh(model.x) |
| arctangent            | atan     | expr = atan(model.x)  |
| hyperbolic arctangent | atanh    | expr = atanh(model.x) |
| cosine                | cos      | expr = cos(model.x)   |
| hyperbolic cosine     | cosh     | expr = cosh(model.x)  |
| exponential           | exp      | expr = exp(model.x)   |
| natural log           | log      | expr = log(model.x)   |
| log base 10           | log10    | expr = log10(model.x) |
| sine                  | sin      | expr = sin(model.x)   |
| square root           | sqrt     | expr = sqrt(model.x)  |
| hyperbolic sine       | sinh     | expr = sinh(model.x)  |
| tangent               | tan      | expr = tan(model.x)   |
| hyperbolic tangent    | tanh     | expr = tanh(model.x)  |

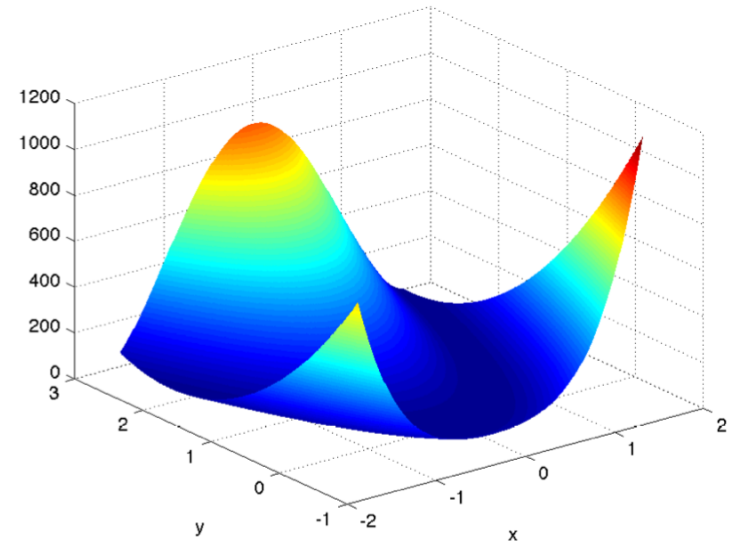
Caution: Always use the intrinsic functions that are part of the Pyomo package.

```
from pyomo.environ import * # imports, e.g., pyomo versions of exp, log, etc.)
from math import * # overrides the pyomo versions with math versions
```

# Example: Rosenbrock function

$$\min_{x,y} f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

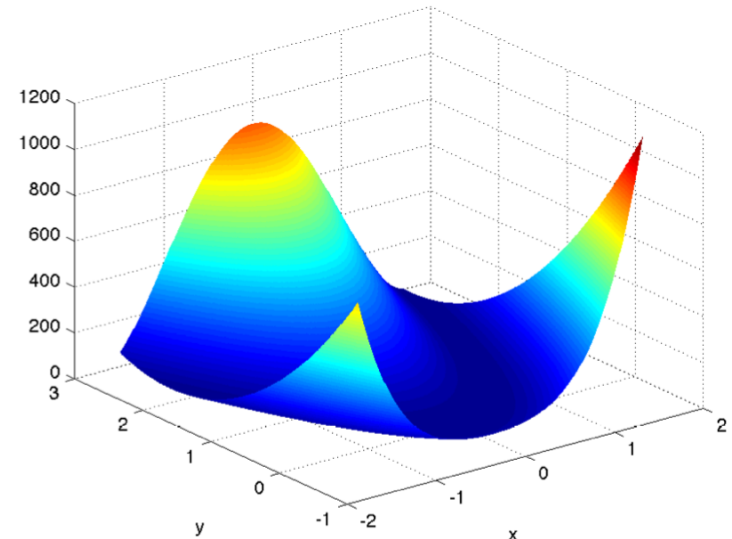
- Minimize the rosenbrock function using Pyomo and IPOPT
- Initialize at  $x=1.5, y=1.5$



# Example: Rosenbrock function

$$\min_{x,y} f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

- Minimize the rosenbrock function using Pyomo and IPOPT
- Initialize at  $x=1.5, y=1.5$



```
rosenbrock.py: A Pyomo model for the Rosenbrock problem
from pyomo.environ import *

model = ConcreteModel()
model.x = Var(initialize=1.5)
model.y = Var(initialize=1.5)

def rosenbrock(m):
 return (1.0-m.x)**2 + 100.0*(m.y - m.x**2)**2
model.obj = Objective(rule=rosenbrock, sense=minimize)

SolverFactory('ipopt').solve(model, tee=True)
model.pprint()
```

**1.1 Alternative Initialization:** Effective initialization can be critical for solving nonlinear problems, since they can have several local solutions and numerical difficulties. Solve the Rosenbrock example using different initial values for the  $x$  variables. Write a loop that varies the initial value from 2.0 to 6.0, solves the problem, and prints the solution for each iteration of the loop. (A solution for this problem can be found in `rosenbrock_init_soln.py`.)

**1.2 Evaluation errors:** Consider the following problem with initial values  $x=5, y=5$ .

$$\begin{aligned} \min_{x,y} \quad & f(x,y) = (x - 1.01)^2 + y^2 \\ \text{s.t.} \quad & y = \sqrt{x - 1.0} \end{aligned}$$

- (a) Starting with `evaluation_error_incomplete.py`, formulate this Pyomo model and solve using IPOPT. You should get a list of errors from the solver. Add the IPOPT solver option `solver.options['halt_on_ampl_error']='yes'` to find the problem. (Hint: error output might be ordered strangely, look up in the console output.) What did you discover? How might you fix this? (A solution for this can be found in `evaluation_error_soln.py`.)
- (b) Add bounds  $x \geq 1$  to fix this problem. Resolve the problem. Comment on the number of iterations and the quality of solution. (Note: The problem still occurs because  $x \geq 1$  is not enforced exactly, and small numerical values still cause the error.) (A solution for this can be found in `evaluation_error_bounds_soln.py`.)
- (c) Think about other solutions for this problem. (e.g.,  $x \geq 1.001$ ). (A solution for this can be found in `evaluation_error_bounds2_soln.py`.)

# Nonlinear: Exercises

**1.3 Alternative Formulations:** Consider the following problem with initial values  $x=5$ ,  $y=5$ .

$$\begin{aligned} \min_{x,y} \quad & f(x,y) = (x - 1.01)^2 + y^2 \\ \text{s.t.} \quad & \frac{x - 1}{y} = 1 \end{aligned}$$

Note that the solution to this problem is  $x=1.005$  and  $y=0.005$ . There are several ways that the problem above can be reformulated. Some examples are shown below. Which ones do you expect to be better? Why? Starting with `formulation_incomplete.py` finish the Pyomo model for each of the formulations and solve with IPOPT. Note the number of iterations and quality of solutions. What can you learn about problem formulation from these examples?

(a) (A solution can be found in `formulation_1_soln.py`.)

$$\begin{aligned} \min_{x,y} \quad & f(x,y) = (x - 1.01)^2 + y^2 \\ \text{s.t.} \quad & \frac{x - 1}{y} = 1 \end{aligned}$$

(b) (A solution for this can be found in `formulation_2_soln.py`.)

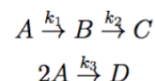
$$\begin{aligned} \min_{x,y} \quad & f(x,y) = (x - 1.01)^2 + y^2 \\ \text{s.t.} \quad & \frac{x}{y + 1} = 1 \end{aligned}$$

(c) (A solution for this can be found in `formulation_3_soln.py`.)

$$\begin{aligned} \min_{x,y} \quad & f(x,y) = (x - 1.01)^2 + y^2 \\ \text{s.t.} \quad & y = x - 1 \end{aligned}$$

(d) Bounds and initialization can be very helpful when solving nonlinear optimization problems. Starting with `formulation_incomplete.py` resolve the original problem, but add bounds,  $y \geq 0$ . Note the number of iterations and quality of solution, and compare with what you found in 1.2 (a). (A solution for this can be found in `formulation_4_soln.py`.)

**1.4 Reactor design problem (Hart et al., 2017; Bequette, 2003):** In this example, we will consider a chemical reactor designed to produce product B from reactant A using a reaction scheme known as the Van de Vusse reaction:



Under appropriate assumptions,  $F$  is the volumetric flowrate through the tank. The concentration of component A in the feed is  $c_{Af}$ , and the concentrations in the reactor are equivalent to the concentrations of each component flowing out of the reactor, given by  $c_A$ ,  $c_B$ ,  $c_C$ , and  $c_D$ .

If the reactor is too small, we will not produce sufficient quantity of B, and if the reactor is too large, much of B will be further reacted to form the undesired product C. Therefore, our goal is to solve for the reactor volume that maximizes the outlet concentration for product B.

The steady-state mole balances for each of the four components are given by,

$$\begin{aligned}0 &= \frac{F}{V}c_{Af} - \frac{F}{V}c_A - k_1c_A - 2k_3c_A^2 \\ 0 &= -\frac{F}{V}c_B + k_1c_A - k_2c_B \\ 0 &= -\frac{F}{V}c_C + k_2c_B \\ 0 &= -\frac{F}{V}c_D + k_3c_A^2\end{aligned}$$

The known parameters for the system are,

$$c_{Af} = 10 \frac{\text{gmol}}{\text{m}^3} \quad k_1 = \frac{5}{6} \text{ min}^{-1} \quad k_2 = \frac{5}{3} \text{ min}^{-1} \quad k_3 = \frac{1}{6000} \frac{\text{m}^3}{\text{mol min}}.$$

Formulate and solve this optimization problem using Pyomo. Since the volumetric flowrate  $F$  always appears as the numerator over the reactor volume  $V$ , it is common to consider this ratio as a single variable, called the space-velocity  $SV$ . (A solution to this problem can be found in `reactor_design_soln.py`.)

# OTHER MATERIAL



# Introduction to IPOPT

$$\begin{array}{ll}
 \min_{\mathbf{x}} & f(\mathbf{x}) \quad \longleftarrow \text{Objective Function} \\
 \text{s.t.} & c(\mathbf{x}) = 0 \quad \longleftarrow \text{Equality Constraints} \\
 & d^L \leq d(\mathbf{x}) \leq d^U \quad \longleftarrow \text{Inequality Constraints} \\
 & x^L \leq \mathbf{x} \leq x^U \quad \longleftarrow \text{Variable Bounds}
 \end{array}$$

$$\mathbf{x} \in \mathbb{R}^n$$

$$f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$$

$$c(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$$

$$d(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^p$$

- $f(\mathbf{x}), c(\mathbf{x}), d(\mathbf{x})$ 
  - ▣ general nonlinear functions (non-convex?)
  - ▣ Smooth ( $C^2$ )
- The  $\mathbf{x}$  variables are continuous
  - ▣  $\mathbf{x}(\mathbf{x}-1)=0$  for discrete conditions really doesn't work

# Introduction to IPOPT

$$\begin{array}{ll}
 \min_{\mathbf{x}} & f(\mathbf{x}) \quad \longleftarrow \text{Cost/Profit, Measure of fit} \\
 \text{s.t.} & c(\mathbf{x}) = 0 \quad \longleftarrow \text{Physics of the system} \\
 & d^L \leq d(\mathbf{x}) \leq d^U \quad \longleftarrow \text{Physical, Performance,} \\
 & x^L \leq \mathbf{x} \leq x^U \quad \longleftarrow \text{Safety Constraints}
 \end{array}$$

$$\mathbf{x} \in \mathbb{R}^n$$

$$f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$$

$$c(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$$

$$d(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^p$$

- $f(\mathbf{x}), c(\mathbf{x}), d(\mathbf{x})$ 
  - ▣ general nonlinear functions (non-convex?)
  - ▣ Smooth ( $C^2$ )
- The  $\mathbf{x}$  variables are continuous
  - ▣  $\mathbf{x}(\mathbf{x}-1)=0$  for discrete conditions really doesn't work

# Large Scale Optimization

## ■ Gradient Based Solution Techniques

$$\begin{array}{ll}
 \min_x & f(x) \\
 \text{s.t.} & c(x) = 0 \\
 & x \geq 0
 \end{array}
 \quad \longrightarrow \quad
 \begin{array}{ll}
 \nabla f(x) + \nabla c(x)^T \cdot \lambda - z = 0 \\
 c(x) = 0 \\
 X \cdot z = 0 \\
 (x \geq 0, z \geq 0)
 \end{array}$$

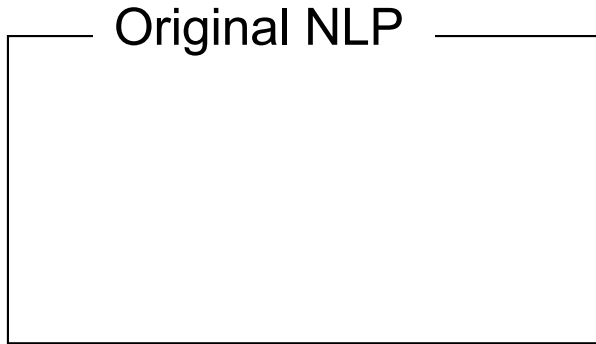
Newton Step

$$\begin{bmatrix} W_k & \nabla c(x_k) \\ \nabla c(x_k)^T & 0 \end{bmatrix} \begin{pmatrix} \Delta x \\ \Delta \lambda \end{pmatrix} = - \begin{bmatrix} \nabla f(x_k) + \nabla c(x_k)^T \lambda_k \\ c(x_k) \end{bmatrix}$$

$$(W_k = \nabla_{xx}^2 \mathcal{L} = \nabla_{xx}^2 f(x_k) + \nabla_{xx}^2 c(x_k) \lambda)$$

Active-set Strategy

# Interior Point Methods



Barrier NLP

$$\begin{aligned} \min_x \quad & f(x) - \mu \cdot \sum_i \ln(x_i) \\ \text{s.t.} \quad & c(x) = 0 \end{aligned}$$

as  $x \rightarrow 0$ ,  $\ln(x) \rightarrow \infty$

as  $\mu \rightarrow 0$ ,  $x^*(\mu) \rightarrow x^*$

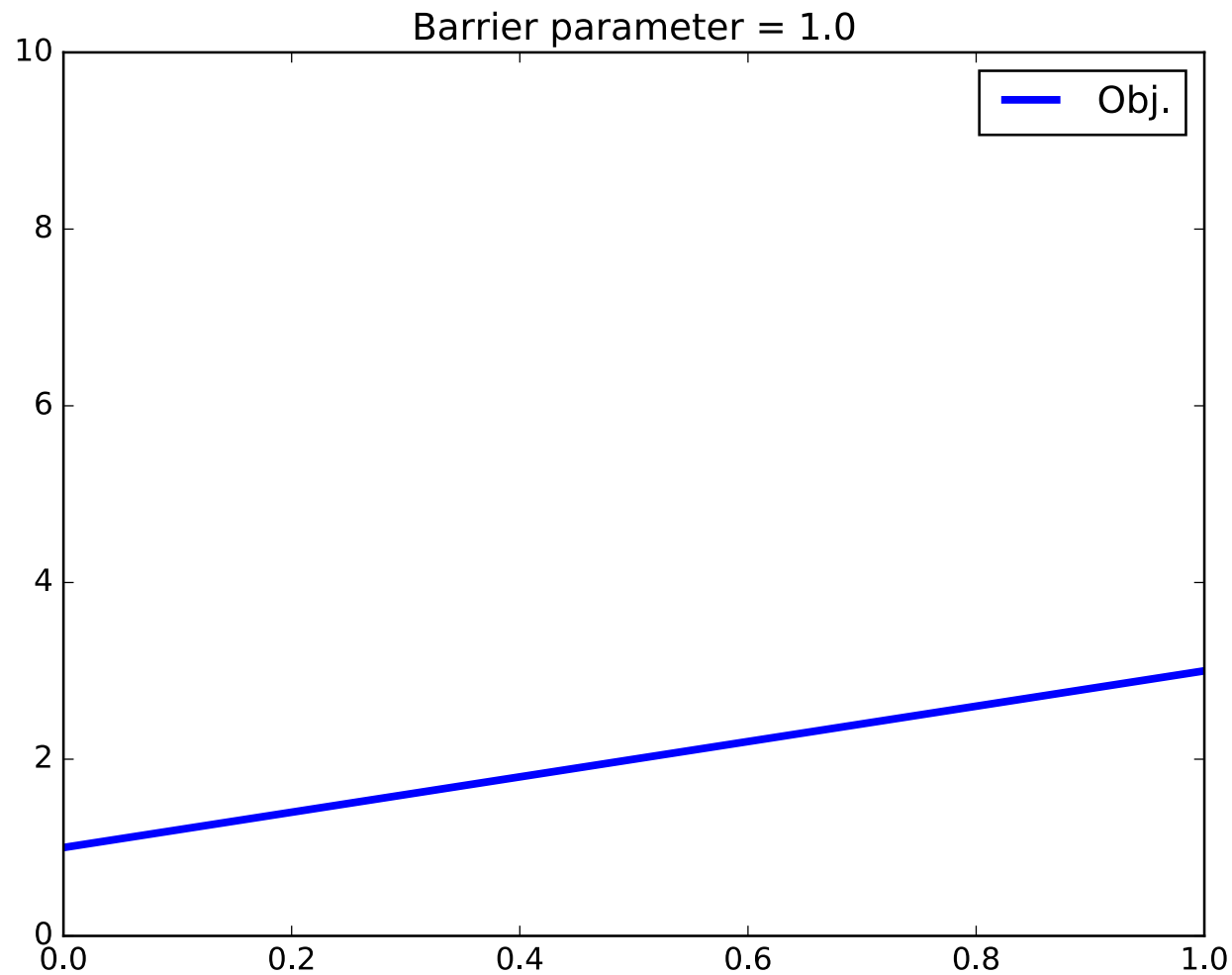
Fiacco & McCormick (1968)

- Initialize  
 $x_0 > 0$ ,  $\mu_0 > 0$ , Set  $l \leftarrow 0$
- Solve Barrier NLP for  $\mu_l$
- Decrease the barrier parameter

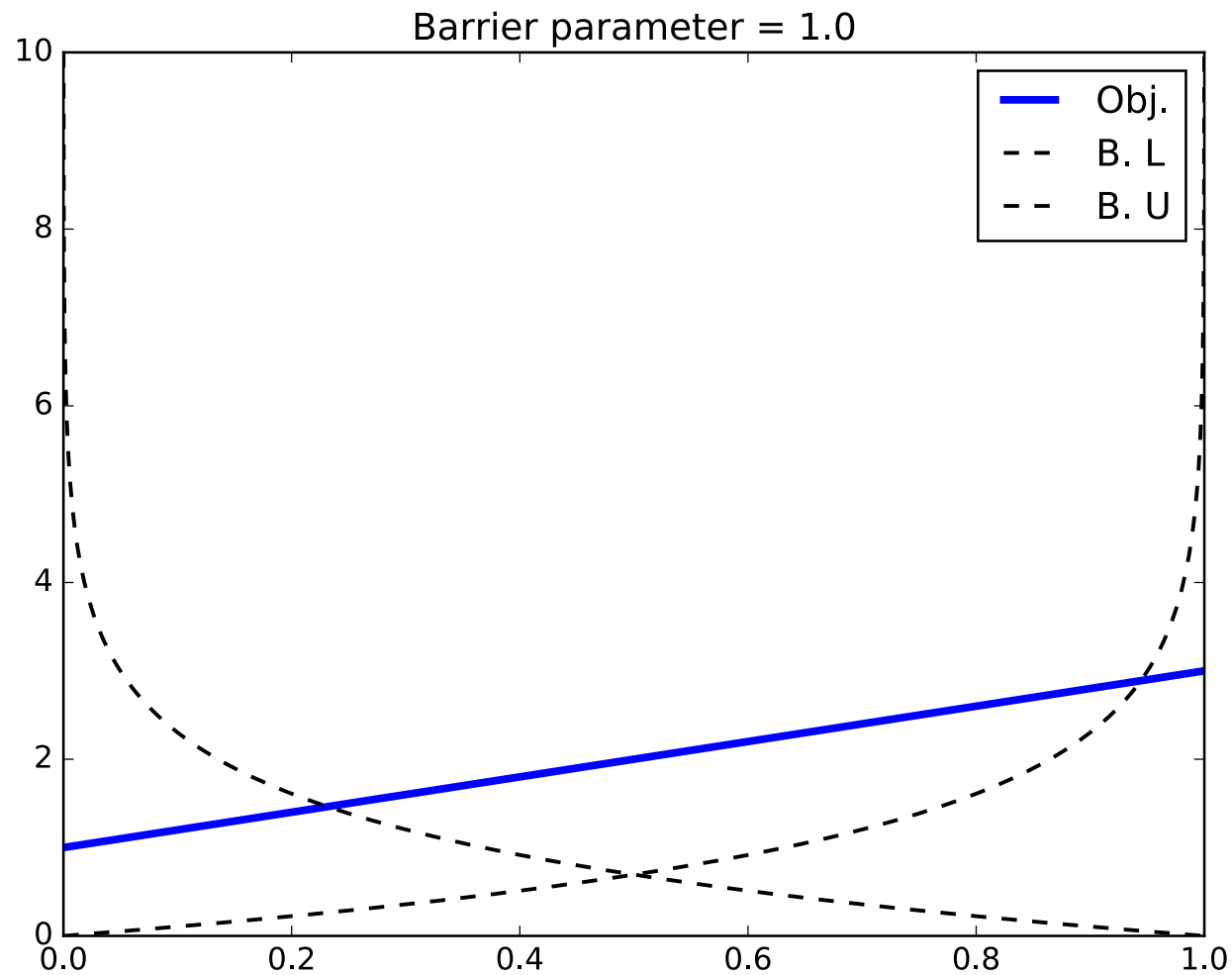
$$\mu_{l+1} < \mu_l$$

- Increase  $l \leftarrow l + 1$
- $$\begin{aligned} f(x) \\ c(x) = 0 \\ x \geq 0 \end{aligned}$$

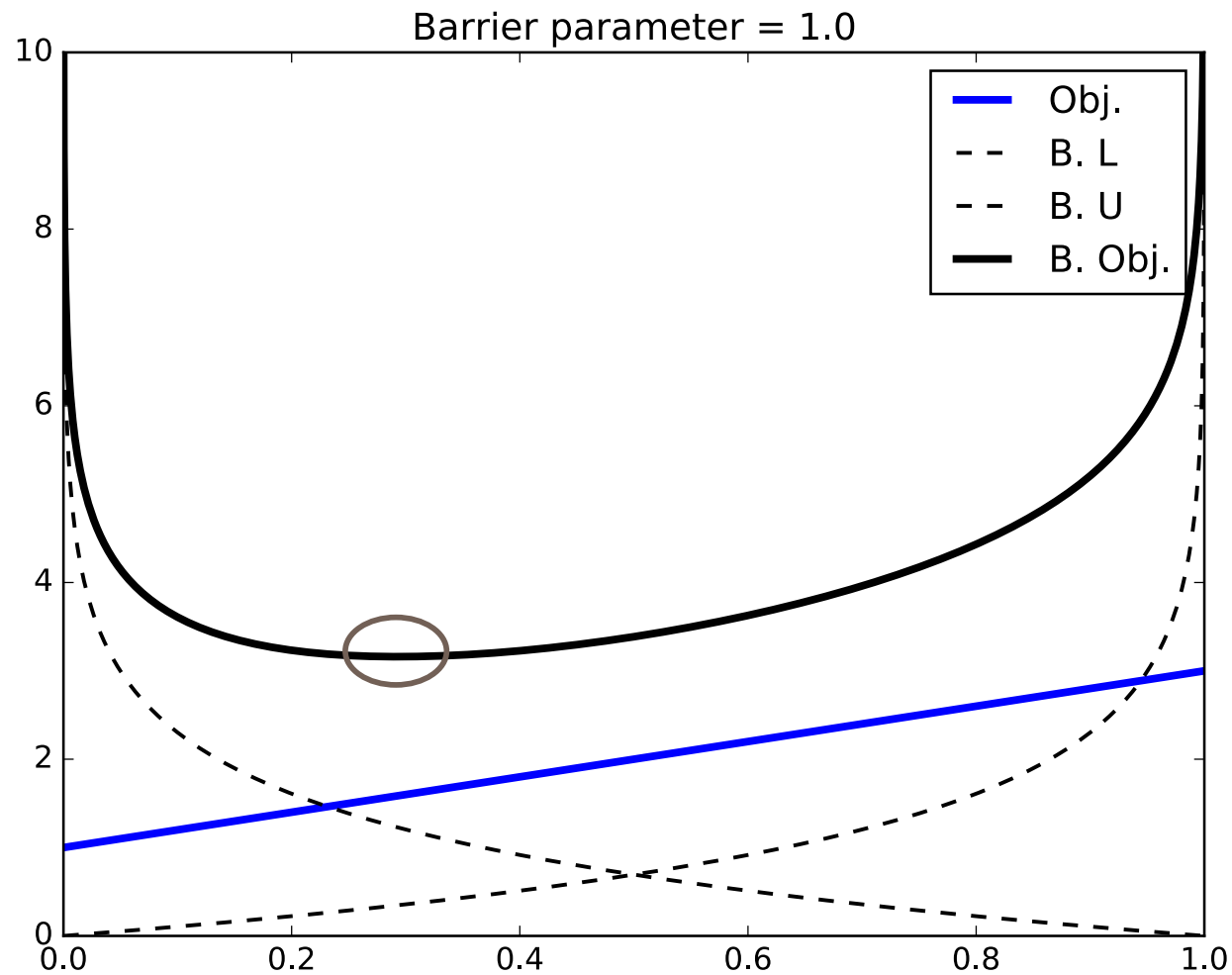
# Effect of Barrier Term



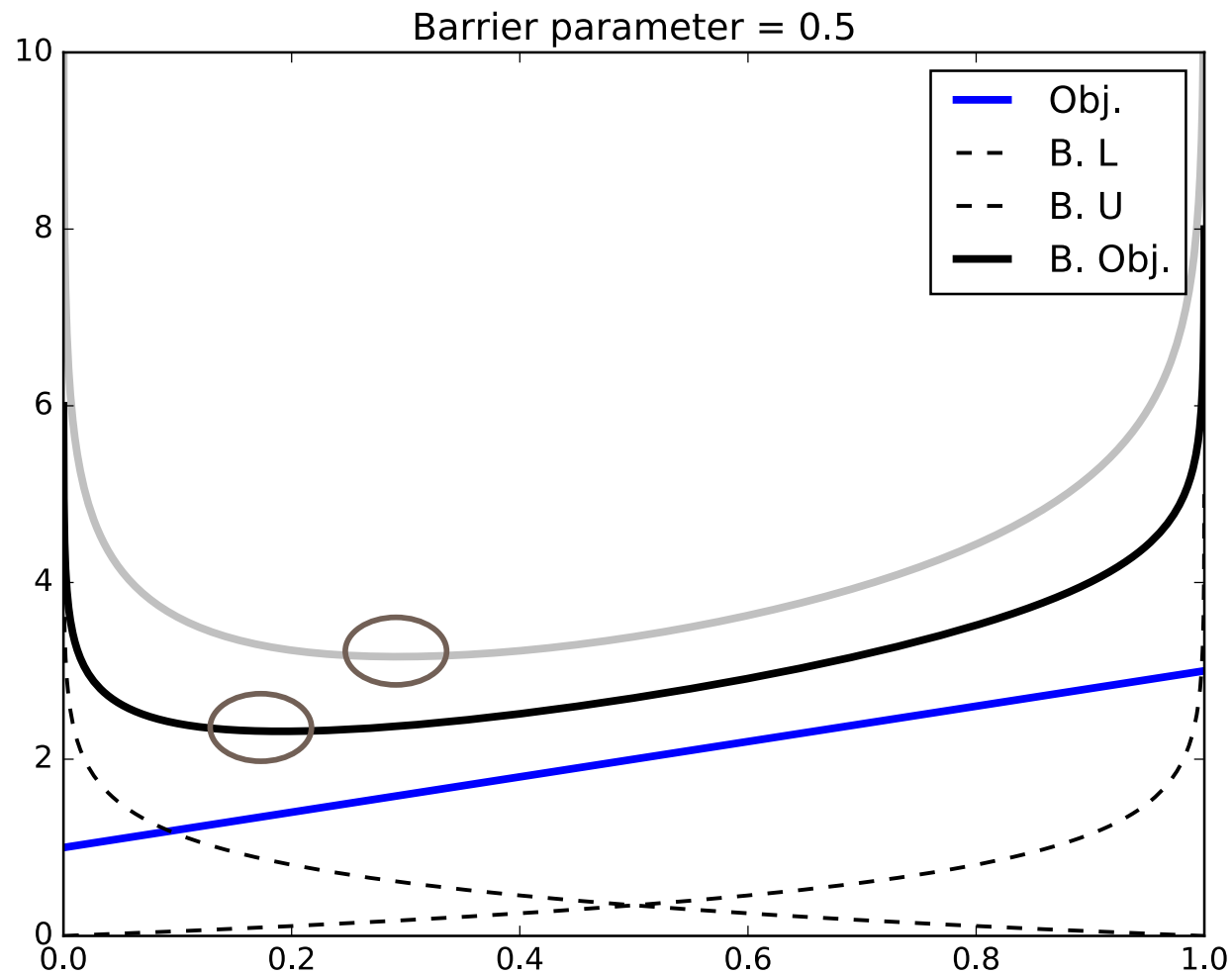
# Effect of Barrier Term



# Effect of Barrier Term

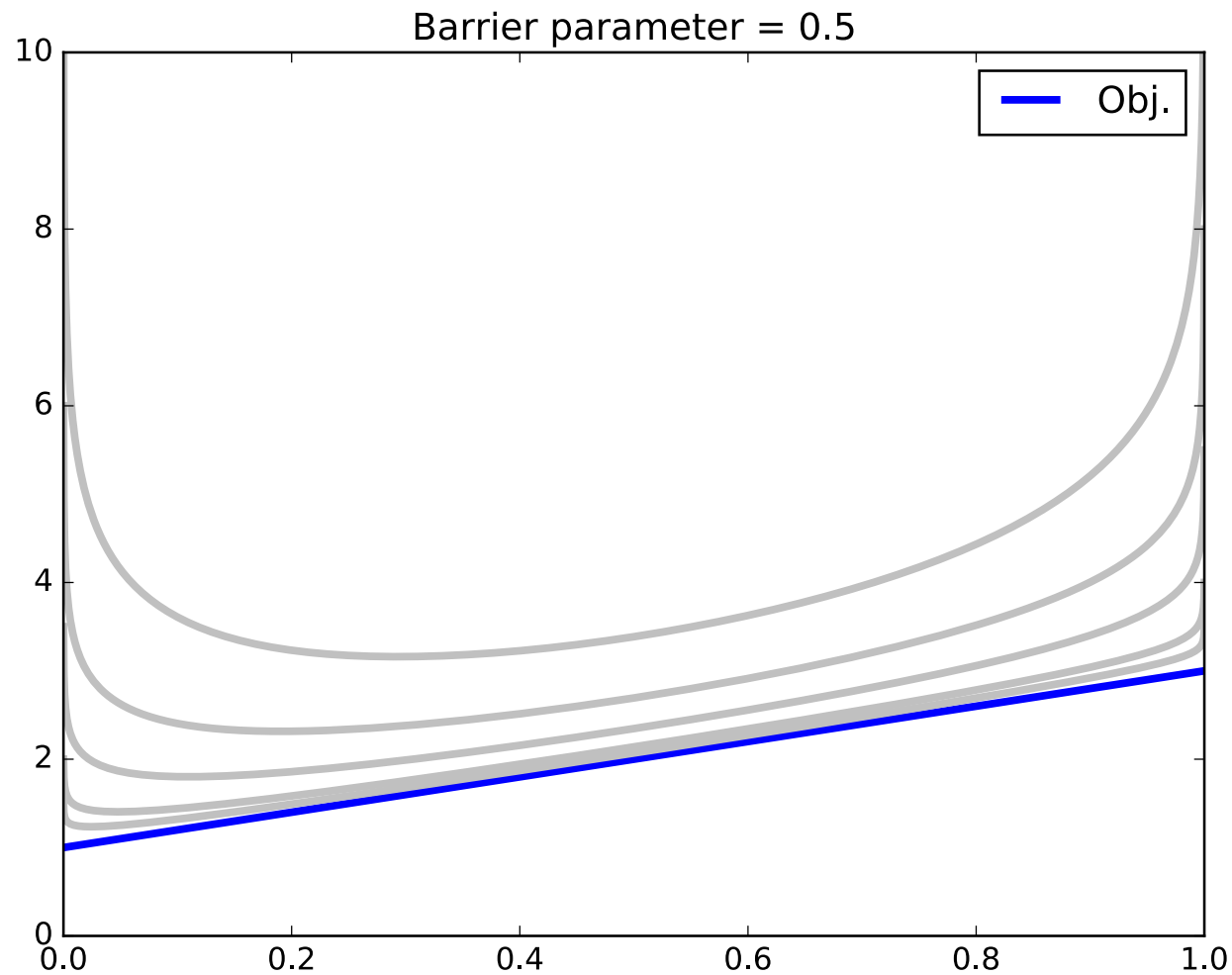


# Effect of Barrier Term





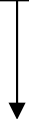
# Effect of Barrier Term



# Interior Point Methods

Original NLP

$$\begin{array}{ll} \min_x & f(x) \\ \text{s.t.} & c(x) = 0 \\ & x \geq 0 \end{array}$$



Barrier NLP

$$\begin{array}{ll} \min_x & \varphi_\mu(x) = f(x) - \mu \cdot \sum_i \ln(x_i) \\ \text{s.t.} & c(x) = 0 \end{array}$$

as  $x \rightarrow 0$ ,  $\ln(x) \rightarrow \infty$

as  $\mu \rightarrow 0$ ,  $x^*(\mu) \rightarrow x^*$

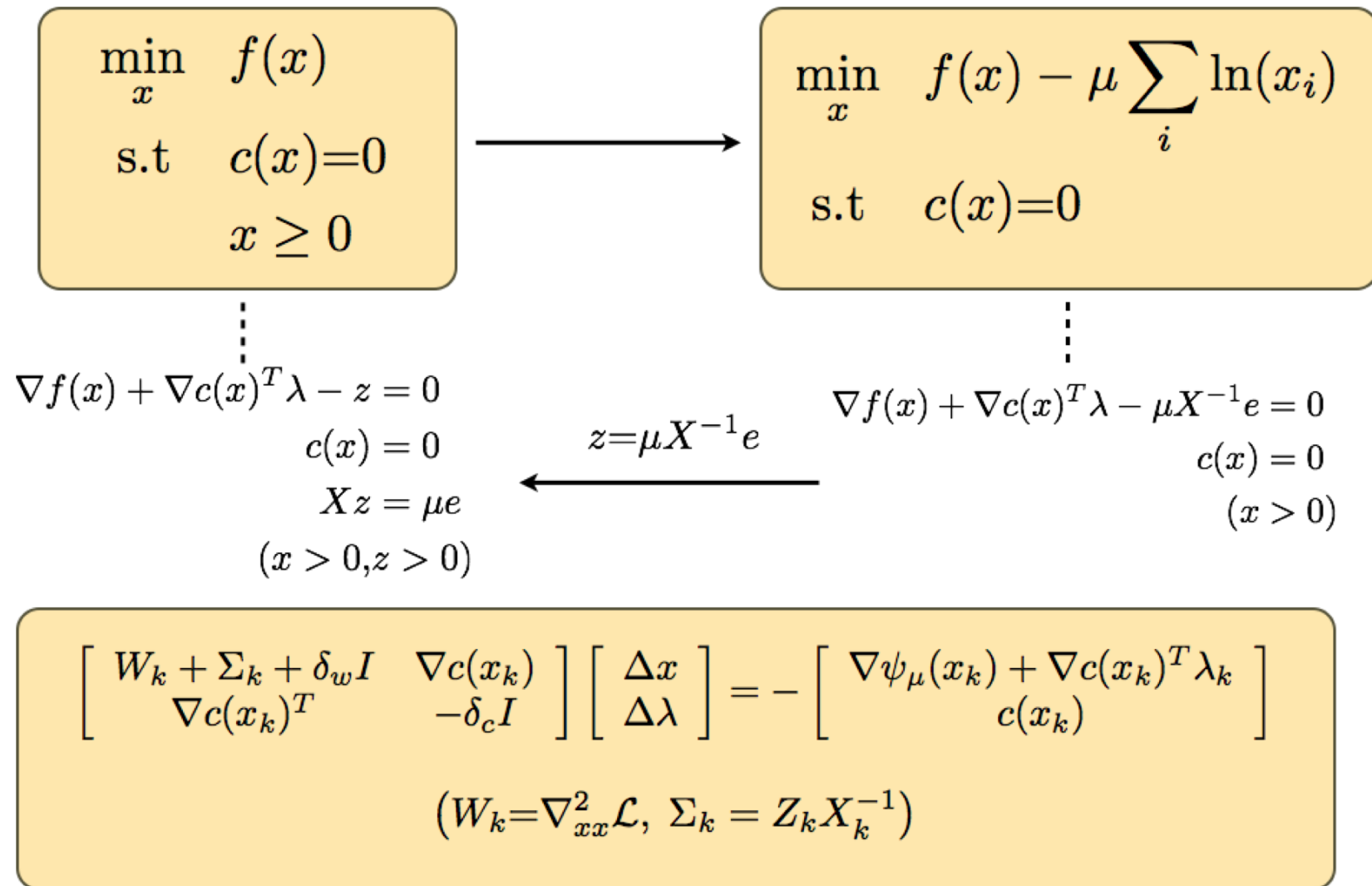
Fiacco & McCormick (1968)

- Initialize  
 $x_0 > 0$ ,  $\mu_0 > 0$ , Set  $l \leftarrow 0$
- Solve Barrier NLP for  $\mu_l$
- Decrease the barrier parameter  
 $\mu_{l+1} < \mu_l$
- Increase  $l \leftarrow l + 1$

Solve Barrier NLP?  
Barrier parameter update?  
Globalization?

- KNITRO (Byrd, Nocedal, Hribar, Waltz)
- LOQO (Benson, Vanderbei, Shanno)
- IPOPT (Wächter, Biegler)

# Interior Point Methods

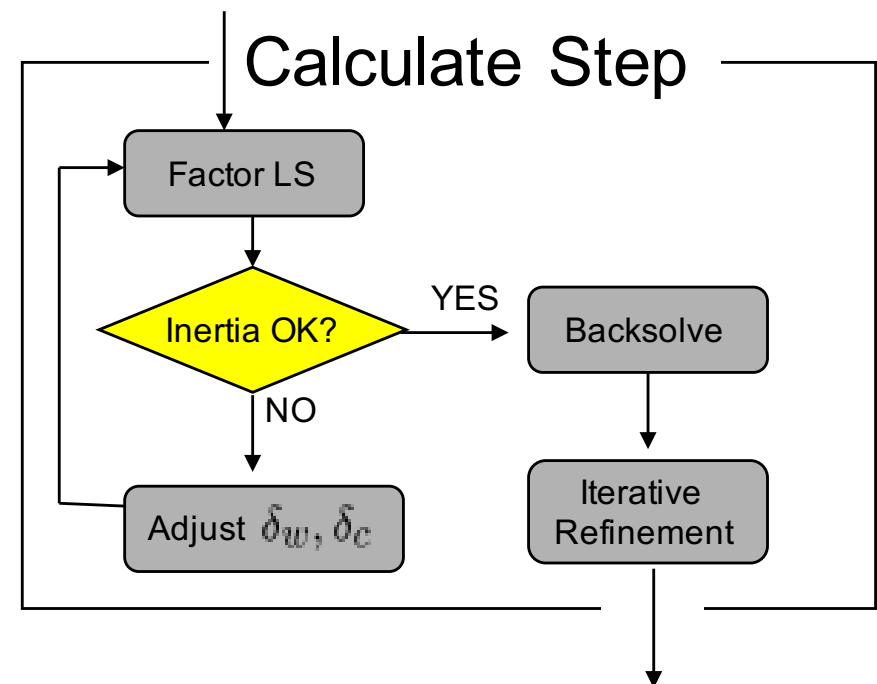
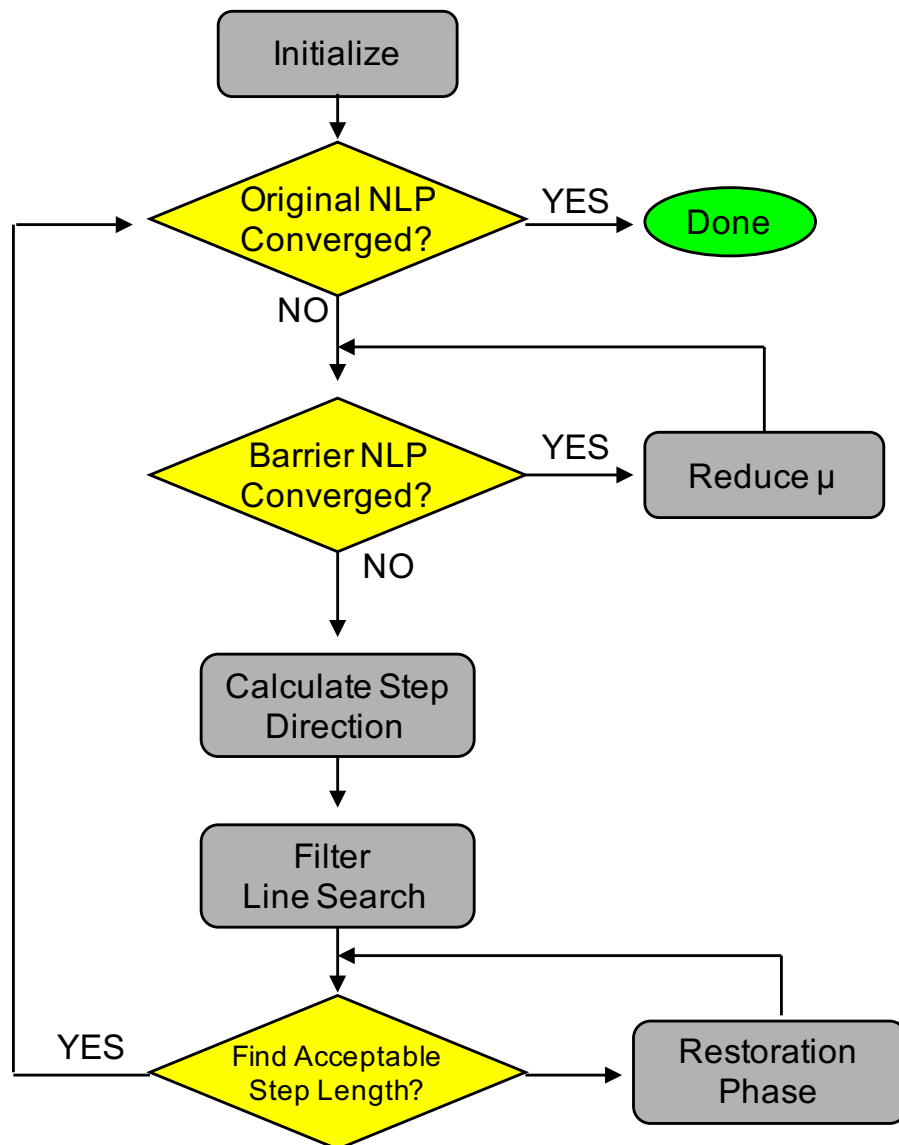


# IPOPT: Other Considerations

---

- Regularization:
  - If certain convexity criteria are not satisfied at a current point, IPOPT may need to regularize. (This can be seen in the output.)
  - We do NOT want to see regularization at the final iteration (solution).
  - Can be an indicator of poor conditioning.
- Globalization:
  - IPOPT uses a filter-based line-search approach
  - Accepts the step if sufficient reduction is seen in objective or constraint violation
- Restoration Phase:
  - Minimize constraint violation
  - Regularized with distance from current point
  - Similar structure to original problem (reuse symbolic factorization)

# IPOPT Algorithm Flowsheet



# IPOPT Output

```

This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Eclipse Public License (EPL).
For more information visit http://projects.coin-or.org/Ipopt

```

This is Ipopt version 3.11.7, running with linear solver ma27.

```
Number of nonzeros in equality constraint Jacobian...: 2
Number of nonzeros in inequality constraint Jacobian.: 0
Number of nonzeros in Lagrangian Hessian.....: 1
```

```
Total number of variables.....: 2
 variables with only lower bounds: 0
 variables with lower and upper bounds: 1
 variables with only upper bounds: 0
Total number of equality constraints.....: 1
Total number of inequality constraints.....: 0
 inequality constraints with only lower bounds: 0
 inequality constraints with lower and upper bounds: 0
 inequality constraints with only upper bounds: 0
```

| iter | objective     | inf_pr   | inf_du   | lg(mu) | d        | lg(rg) | alpha_du | alpha_pr | ls |
|------|---------------|----------|----------|--------|----------|--------|----------|----------|----|
| 0    | 5.0000000e-01 | 2.50e-01 | 5.00e-01 | -1.0   | 0.00e+00 | -      | 0.00e+00 | 0.00e+00 | 0  |

# IPOPT Output

| iter | objective      | inf_pr   | inf_du   | lg(mu) | d        | lg(rg) | alpha_du | alpha_pr  | ls |
|------|----------------|----------|----------|--------|----------|--------|----------|-----------|----|
| 0    | 5.0000000e-01  | 2.50e-01 | 5.00e-01 | -1.0   | 0.00e+00 | -      | 0.00e+00 | 0.00e+00  | 0  |
| 1    | 2.4298076e-01  | 2.33e-01 | 7.67e-01 | -1.0   | 5.20e-01 | -      | 7.73e-01 | 9.52e-01h | 1  |
| 2    | 2.6898113e-02  | 7.23e-05 | 4.09e-04 | -1.7   | 2.16e-01 | -      | 1.00e+00 | 1.00e+00h | 1  |
| 3    | 1.8655807e-04  | 1.83e-04 | 7.86e-05 | -3.8   | 2.68e-02 | -      | 1.00e+00 | 9.97e-01f | 1  |
| 4    | 1.8250072e-06  | 1.23e-12 | 2.22e-16 | -5.7   | 1.85e-04 | -      | 1.00e+00 | 1.00e+00h | 1  |
| 5    | -1.7494097e-08 | 8.48e-13 | 0.00e+00 | -8.6   | 1.84e-06 | -      | 1.00e+00 | 1.00e+00h | 1  |

Number of Iterations.....: 5

|                            | (scaled)                 | (unscaled)               |
|----------------------------|--------------------------|--------------------------|
| Objective.....:            | -1.7494096510394117e-08  | -1.7494096510394117e-08  |
| Dual infeasibility.....:   | 0.000000000000000000e+00 | 0.000000000000000000e+00 |
| Constraint violation.....: | 8.4843243541854463e-13   | 8.4843243541854463e-13   |
| Complementarity.....:      | 2.5050549017950606e-09   | 2.5050549017950606e-09   |
| Overall NLP error.....:    | 2.5050549017950606e-09   | 2.5050549017950606e-09   |

- iter: iterations (codes)
- objective: objective
- Inf\_pr: primal infeasibility (constraints satisfied? current constraint violation)
- Inf\_du: dual infeasibility (am I optimal?)
- lg(mu): log of the barrier parameter, mu
- ||d||: length of the current stepsize
- lg(rg): log of the regularization parameter
- alpha\_du: stepsize for dual variables
- alpha\_pr: stepsize for primal variables
- ls: number of line-search steps

# Exit Conditions

---

- Successful Exit
- Successful Exit with regularization at solution
- Infeasible
- Unbounded



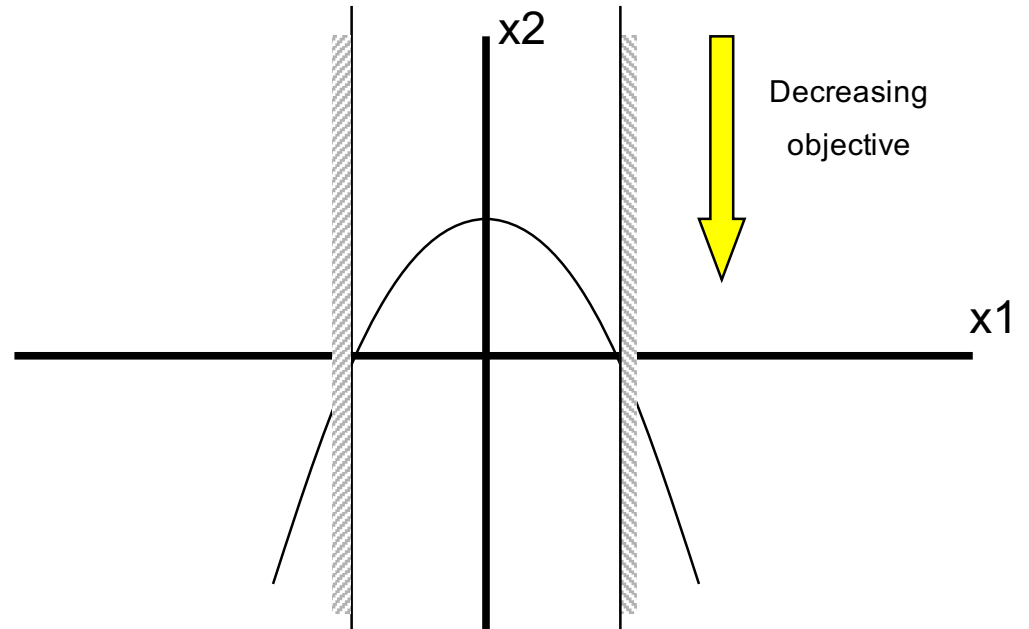
# Exit Conditions: Successful

$$\min x_2$$

$$-x_2 = x_1^2 - 1$$

$$-1 \leq x_1 \leq 1$$

Initialize at  $(x_1=0.5, x_2=0.5)$



# Exit Conditions: Successful

```

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
 0 5.0000000e-01 2.50e-01 5.00e-01 -1.0 0.00e+00 - 0.00e+00 0.00e+00 0
 1 2.4298076e-01 2.33e-01 7.67e-01 -1.0 5.20e-01 - 7.73e-01 9.52e-01h 1
 2 2.6898113e-02 7.23e-05 4.09e-04 -1.7 2.16e-01 - 1.00e+00 1.00e+00h 1
 3 1.8655807e-04 1.83e-04 7.86e-05 -3.8 2.68e-02 - 1.00e+00 9.97e-01f 1
 4 1.8250072e-06 1.23e-12 2.22e-16 -5.7 1.85e-04 - 1.00e+00 1.00e+00h 1
 5 -1.7494097e-08 8.48e-13 0.00e+00 -8.6 1.84e-06 - 1.00e+00 1.00e+00h 1

```

Number of Iterations.....: 5

|                            | (scaled)                | (unscaled)              |
|----------------------------|-------------------------|-------------------------|
| Objective.....:            | -1.7494096510367012e-08 | -1.7494096510367012e-08 |
| Dual infeasibility.....:   | 0.0000000000000000e+00  | 0.0000000000000000e+00  |
| Constraint violation.....: | 8.4843243541854463e-13  | 8.4843243541854463e-13  |
| Complementarity.....:      | 2.5050549017950606e-09  | 2.5050549017950606e-09  |
| Overall NLP error.....:    | 2.5050549017950606e-09  | 2.5050549017950606e-09  |

. . .

EXIT: Optimal Solution Found.

Ipopt 3.11.1: Optimal Solution Found

```

*** soln
x1 = 1.0
x2 = -1.7494096510367012e-08

```

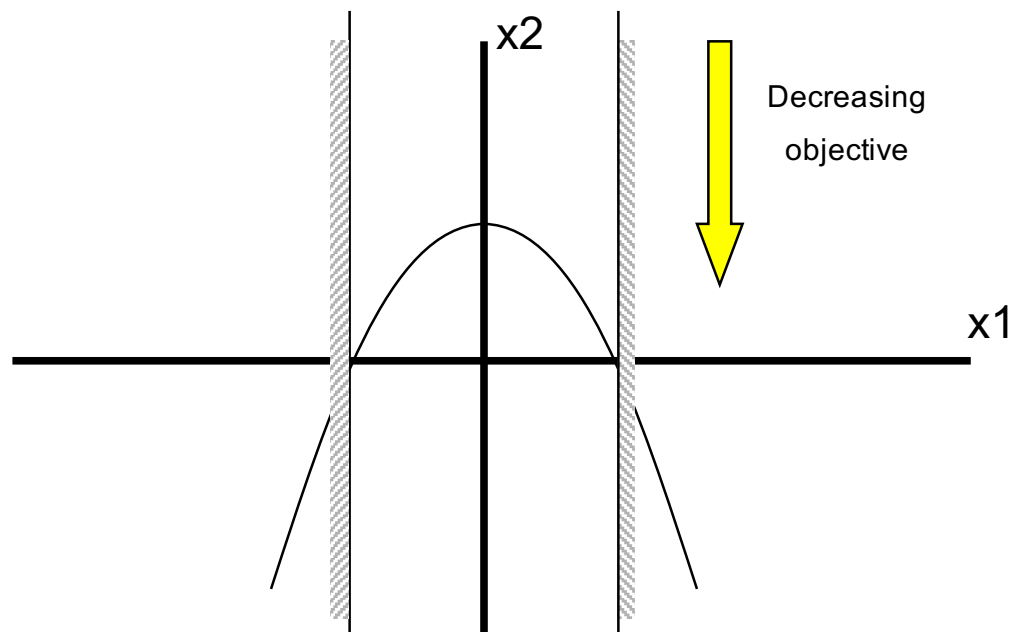
# Exit Conditions: Successful w/ Regularization

$$\min x_2$$

$$-x_2 = x_1^2 - 1$$

$$-1 \leq x_1 \leq 1$$

Initialize at  $(x_1=0.0, x_2=2.0)$



# Exit Conditions: Successful w/ Regularization

```

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
 0 2.0000000e+00 1.00e+00 0.00e+00 -1.0 0.00e+00 - 0.00e+00 0.00e+00 0
 1 1.0000000e+00 0.00e+00 1.00e-04 -1.7 1.00e+00 -4.0 1.00e+00 1.00e+00h 1
 2 1.0000000e+00 0.00e+00 0.00e+00 -3.8 0.00e+00 0.9 1.00e+00 1.00e+00 0
 3 1.0000000e+00 0.00e+00 0.00e+00 -5.7 0.00e+00 0.5 1.00e+00 1.00e+00T 0
 4 1.0000000e+00 0.00e+00 0.00e+00 -8.6 0.00e+00 0.9 1.00e+00 1.00e+00T 0

```

Number of Iterations.....: 4

|                            | (scaled)               | (unscaled)             |
|----------------------------|------------------------|------------------------|
| Objective.....:            | 1.0000000000000000e+00 | 1.0000000000000000e+00 |
| Dual infeasibility.....:   | 0.0000000000000000e+00 | 0.0000000000000000e+00 |
| Constraint violation.....: | 0.0000000000000000e+00 | 0.0000000000000000e+00 |
| Complementarity.....:      | 2.5059035596800808e-09 | 2.5059035596800808e-09 |
| Overall NLP error.....:    | 2.5059035596800808e-09 | 2.5059035596800808e-09 |

. . .

EXIT: Optimal Solution Found.

Ipopt 3.11.1: Optimal Solution Found

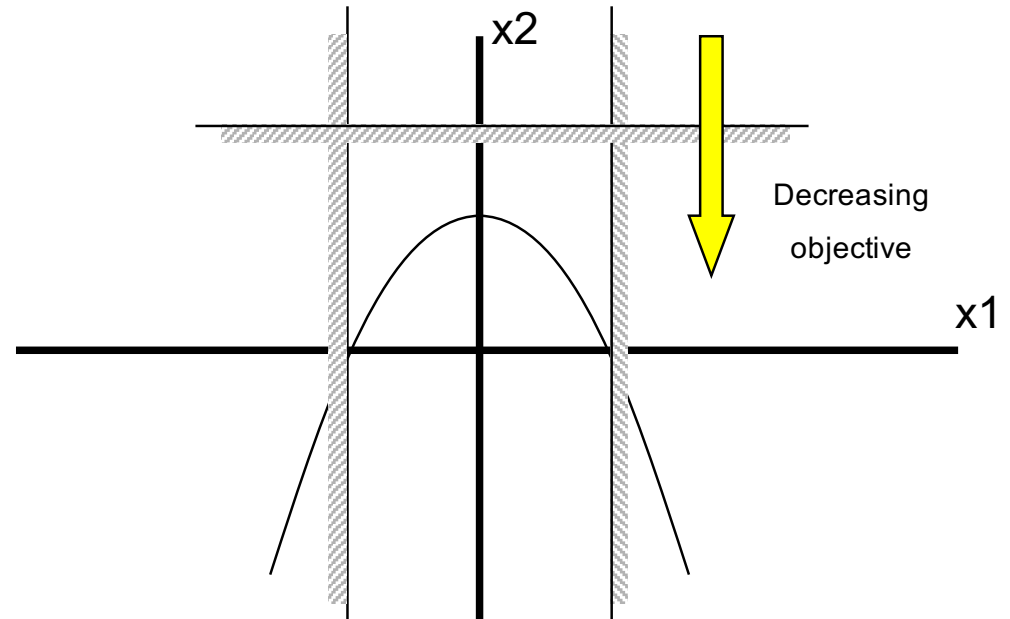
```

*** soln
x1 = 0.0
x2 = 1.0

```

# Exit Conditions: Infeasible

$$\begin{array}{ll}\min & -x_2 \\ \text{s.t.} & -x_2 = x_1^2 - 1 \\ & -1 \leq x_1 \leq 1 \\ & x_2 \geq 2\end{array}$$



# Exit Conditions: Infeasible

```

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
. . . .
 5 2.0000016e+00 1.00e+00 4.74e+03 -1.0 3.47e+01 - 2.79e-02 4.16e-04h 7
 6r 2.0000016e+00 1.00e+00 1.00e+03 0.0 0.00e+00 - 0.00e+00 4.44e-07R 3
 7r 2.0010000e+00 1.01e+00 1.74e+02 0.0 8.73e-02 - 1.00e+00 1.00e+00f 1
 8r 2.0010010e+00 1.00e+00 1.32e-03 0.0 8.73e-02 - 1.00e+00 1.00e+00f 1
 9r 2.0000080e+00 1.00e+00 5.18e-03 -2.1 6.12e-03 - 9.94e-01 9.99e-01h 1
iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
10r 2.0000000e+00 1.00e+00 3.73e-06 -4.7 7.99e-06 - 1.00e+00 1.00e+00f 1
11r 2.0000000e+00 1.00e+00 1.79e-07 -7.1 2.85e-08 - 1.00e+00 1.00e+00f 1

```

Number of Iterations.....: 11

|                            | (scaled)               | (unscaled)             |
|----------------------------|------------------------|------------------------|
| Objective.....:            | 1.9999999800009090e+00 | 1.9999999800009090e+00 |
| Dual infeasibility.....:   | 1.0000000002321485e+00 | 1.0000000002321485e+00 |
| Constraint violation.....: | 9.9999998000090895e-01 | 9.9999998000090895e-01 |
| Complementarity.....:      | 9.0909091652062654e-10 | 9.0909091652062654e-10 |
| Overall NLP error.....:    | 9.9999998000090895e-01 | 1.0000000002321485e+00 |

. . .

EXIT: Converged to a point of local infeasibility. Problem may be infeasible.

Ipopt 3.11.1: Converged to a locally infeasible point. Problem may be infeasible.

WARNING - Loading a SolverResults object with a warning status into model=unknown; message from solver=Ipopt 3.11.1\x3a Converged to a locally infeasible point. Problem may be infeasible.

\*\*\* soln

x1 = -6.353194883662875e-12

x2 = 2.0

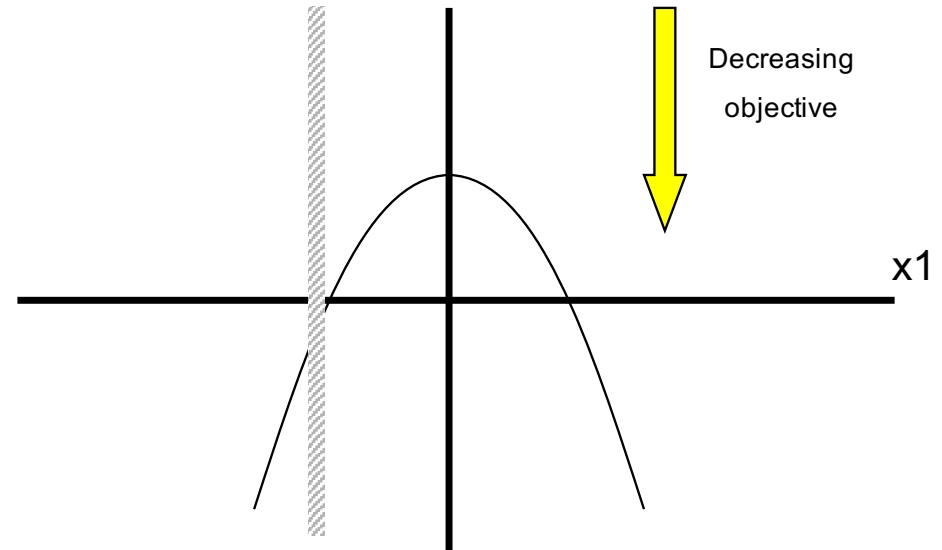
# Exit Conditions: Unbounded

$$\min -x_2$$

$$\text{s.t.} \quad -x_2 = x_1^2 - 1$$

$$-1 \leq x_1$$

Initialize at  $(x_1=0.5, x_2=0.5)$



# Exit Conditions: Unbounded

```

iter objective inf_pr inf_du lg(mu) ||d|| lg(rg) alpha_du alpha_pr ls
. . . .
 45 -2.2420218e+11 1.00e+04 1.00e+00 -1.7 1.25e+19 -19.1 3.55e-08 7.11e-15f 48
 46 -2.2420225e+11 1.00e+04 1.00e+00 -1.7 3.75e+19 -19.6 1.20e-08 1.78e-15f 50
 47 -2.2420229e+11 1.00e+04 1.00e+00 -1.7 1.25e+19 -19.1 1.00e+00 3.55e-15f 49
 48 -3.7503956e+19 1.57e+27 8.36e+09 -1.7 3.75e+19 -19.6 1.18e-08 1.00e+00w 1
 49 -1.3750923e+20 3.92e+26 2.09e+09 -1.7 1.00e+20 -20.0 1.00e+00 1.00e+00w 1

```

Number of Iterations.....: 49

|                            | (scaled)                | (unscaled)              |
|----------------------------|-------------------------|-------------------------|
| Objective.....:            | -1.3750923074037683e+20 | -1.3750923074037683e+20 |
| Dual infeasibility.....:   | 2.0888873315629249e+09  | 2.0888873315629249e+09  |
| Constraint violation.....: | 3.9209747283936173e+26  | 3.9209747283936173e+26  |
| Complementarity.....:      | 3.1115099971882619e+03  | 3.1115099971882619e+03  |
| Overall NLP error.....:    | 3.9209747283936173e+26  | 3.9209747283936173e+26  |

EXIT: Iterates diverging; problem might be unbounded.

Ipopt 3.11.1: Iterates diverging; problem might be unbounded.

WARNING - Loading a SolverResults object with a warning status into model=unknown; message from solver=Ipopt 3.11.1\x3a Iterates diverging; problem might be unbounded.

```

*** soln
x1 = 0.5
x2 = 0.5

```



# IPOPT Options

---

- Solver options can be set through scripts (and the pyomo command line)
- `print_options_documentation` yes
  - Outputs the complete set of IPOPT options (with documentation and their defaults)
- `mu_init`
  - Sets the initial value of the barrier parameter
  - Can be helpful to make this smaller when initial guesses are known to be good
- `bounds_push`
  - By default, IPOPT pushes the bounds a little further out.
  - This can be set to remove this behavior
  - E.g., `sqrt(x)`,  $x \geq 0$
- `linear_solver`
  - Set the linear solver that will be used for the KKT system
  - Significantly better performance with HSL (MA27) over default MUMPS
- `print_user_options`
  - Print options set and whether or not they were used
  - Helpful to detect mismatched options

# IPOPT Options

```
rosenbrock_options.py: A Pyomo model for the Rosenbrock problem
from pyomo.environ import *

model = ConcreteModel()
model.x = Var()
model.y = Var()

def rosenbrock(m):
 return (1.0-m.x)**2 + 100.0*(m.y - m.x**2)**2
model.obj = Objective(rule=rosenbrock, sense=minimize)

solver = SolverFactory('ipopt')
solver.options['mu_init'] = 1e-4
solver.options['print_user_options'] = 'yes'
solver.options['ma27_pivtol'] = 1e-4
solver.solve(model, tee=True)

print()
print('*** Solution *** :')
print('x:', value(model.x))
print('y:', value(model.y))
```

# IPOPT Options

```
ma27_pivtol=0.0001
print_user_options=yes
mu_init=0.0001
ma27_pivtol=0.0001
print_user_options=yes
mu_init=0.0001
```

List of user-set options:

| Name               | Value    | used |
|--------------------|----------|------|
| ma27_pivtol        | = 0.0001 | no   |
| mu_init            | = 0.0001 | yes  |
| print_user_options | = yes    | yes  |

```

This program contains Ipopt, a library for large-scale nonlinear optimization.
Ipopt is released as open source code under the Eclipse Public License (EPL).
For more information visit http://projects.coin-or.org/Ipopt

```

NOTE: You are using Ipopt by default with the MUMPS linear solver.  
Other linear solvers might be more efficient (see Ipopt documentation).

This is Ipopt version 3.11.1, running with linear solver mumps.

# Other Considerations

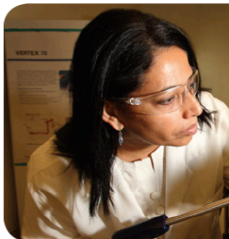
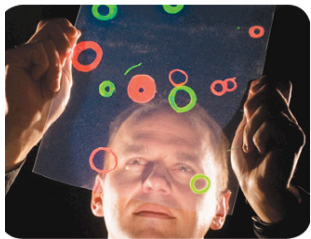
---

- Linear Solvers
  - Performance of IPOPT is HIGHLY dependent on the linear solver selected
  - The version of IPOPT installed in this workshop uses a freely distributable linear solver, MUMPS.
  - Performance of IPOPT with the Harwell Subroutine Library suite can be significantly better
  - MA27 (HSL) is **free** for personal and commercial use, but not **distributable**
  - Recommend downloading and installing MA27 from HSL
- Variable Initialization
  - Proper initialization of nonlinear problems can be critical for effective solution.
  - Strategies include:
    - Using understood physics or past successful solutions
    - Solving simpler problem(s) first, progressing to more difficult

# Other Considerations

---

- Undefined Evaluations
  - Many mathematical functions have a valid domain, and evaluation outside that domain causes errors
  - Add appropriate bounds to variables to keep them inside valid domain
  - Note that solvers use first and second derivatives. While  $\sqrt{x}$  is valid at  $x=0$ ,  $1/\sqrt{x}$  is not
- Problem Scaling
  - Scale model to avoid variables, constraints, derivatives with different scales.
- Formulation Matters



Sandia  
National  
Laboratories

*Exceptional  
service  
in the  
national  
interest*

## 6. Structured Modeling & Transformations



U.S. DEPARTMENT OF  
**ENERGY**



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Is this an *optimization model*?

---

$$\begin{array}{ll}\min & c^T x \\ \text{s. t.} & Ax \leq b \\ & x \in \mathbb{R}^n\end{array}$$

# Models are for *Modelers*

$$\begin{array}{ll} \min & c^T x \\ \text{s. t.} & Ax \leq b \\ & x \in \mathbb{R}^n \end{array}$$

- I would argue this is an *optimization problem*!
- So, what's a *model*?
  - A general representation of a class of problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
  - Incorporates assumptions and simplifications
  - Is both tractable and valid
    - (although these are often contradictory goals)



# Models are for *Modelers*

$$\begin{array}{ll} \min & c^T x \\ \text{s. t.} & Ax \leq b \\ & x \in \mathbb{R}^n \end{array}$$

- I would argue this is an *optimization problem*!
- So, what's a *model*?
  - A general representation of a class of problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
  - Incorporates assumptions and simplifications
  - Is both tractable and valid
    - (although these are often contradictory goals)

# Optimization problems: Model instances



- We seldom have a single *problem* to solve
  - Rather we would like to write a *single model* for a *class of problems*
    - Key design feature of many AMLs (e.g. strongly encouraged by AMPL)
  - Why?
    - Test small, deploy big
    - Tomorrow's problem is different from today's
    - Data may be
      - Huge
      - Machine-generated
      - Stored externally (loaded from external tools, e.g. databases)

# Models are for *Modelers*

$$\begin{array}{ll}\min & c^T x \\ \text{s.t.} & Ax \leq b \\ & x \in \mathbb{R}^n\end{array}$$

- I would argue this is an *optimization problem*!
- So, what's a *model*?
  - A general representation of a class of problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
  - Incorporates assumptions and simplifications
  - Is both tractable and valid
    - (although these are often contradictory goals)

# What is *model structure*?

$$\begin{array}{ll} \min & c^T x \\ \text{s. t.} & Ax \leq b \\ & x \in \mathbb{R}^n \end{array}$$

- Unlike a solver, modelers don't think in terms of “ $A$ ”
  - Rather, I think in terms of repeated (indexed) units
    - Sets (1-, 2-, n- dimensional)
    - Vectors or matrices of variables
    - Groups of related constraints (blocks)
- The model may not be “flat”
  - Block diagonal (e.g., scenarios in stochastic programming)
  - Graph-based (e.g., network flow)
  - Hierarchically defined (e.g., a model composed of sub-models)

# Models are for *Modelers*

$$\begin{array}{ll}\min & c^T x \\ \text{s. t.} & Ax \leq b \\ & x \in \mathbb{R}^n\end{array}$$

- I would argue this is an *optimization problem*!
- So, what's a *model*?
  - A general representation of a class of problems
    - Data (instance) independent
  - Represents the modeler's understanding of the class of problems
    - Explicitly annotates and conveys the class structure
  - Incorporates assumptions and simplifications
  - Is both tractable and valid
    - (although these are often contradictory goals)

- The “highest fidelity” model of a system is rarely tractable
  - Delicate balance between the model we want to solve and the solver we want to use
  - What can we do?
    - Simplify (reduce the model scope)
    - Approximate (relax or recast constraints)
    - Iterate (solve a series of related problems to develop the solution to the original problem)
- Optimization 101 ingrains this tension into us; consider:

$$\begin{array}{ll} \max & \text{abs}(x - 3) \\ \text{s.t.} & [\dots] \end{array}$$

# “Modeling” absolute value

- This probably makes you cringe:
  - “Experienced modelers would never write `abs ( )`!”
- Instead, we write:

$$\begin{array}{ll} \max & \text{abs}(x - 3) \\ \text{s.t.} & [\dots] \end{array}$$

$$\begin{array}{ll} \max & \text{abs}X \\ \text{s.t.} & \text{abs}X = \text{neg}X + \text{pos}X \\ & \text{neg}X \leq My \\ & \text{pos}X \leq M(1 - y) \\ & X - 3 = \text{pos}X - \text{neg}X \\ & \text{pos}X \geq 0, \text{neg}X \geq 0 \\ & y \in \{0, 1\} \\ & [\dots] \end{array}$$

# “Modeling” absolute value

- This probably makes you cringe:
  - “Experienced modelers would never write `abs ( )`!”

$$\begin{array}{ll} \max & abs(x-3) \\ s.t. & [...] \end{array}$$

- Instead, we write:
- But what if “[...]” is a nonlinear model? Then,

$$\begin{aligned} absX &= \sqrt{x^2 + \varepsilon} \\ absX &= \frac{2x}{1 + e^{-x/h}} - x \end{aligned}$$

$$\begin{array}{ll} \max & absX \\ s.t. & absX = negX + posX \\ & negX \leq My \\ & posX \leq M(1-y) \\ & X-3 = posX - negX \\ & posX \geq 0, negX \geq 0 \\ & y \in \{0,1\} \\ & [...] \end{array}$$

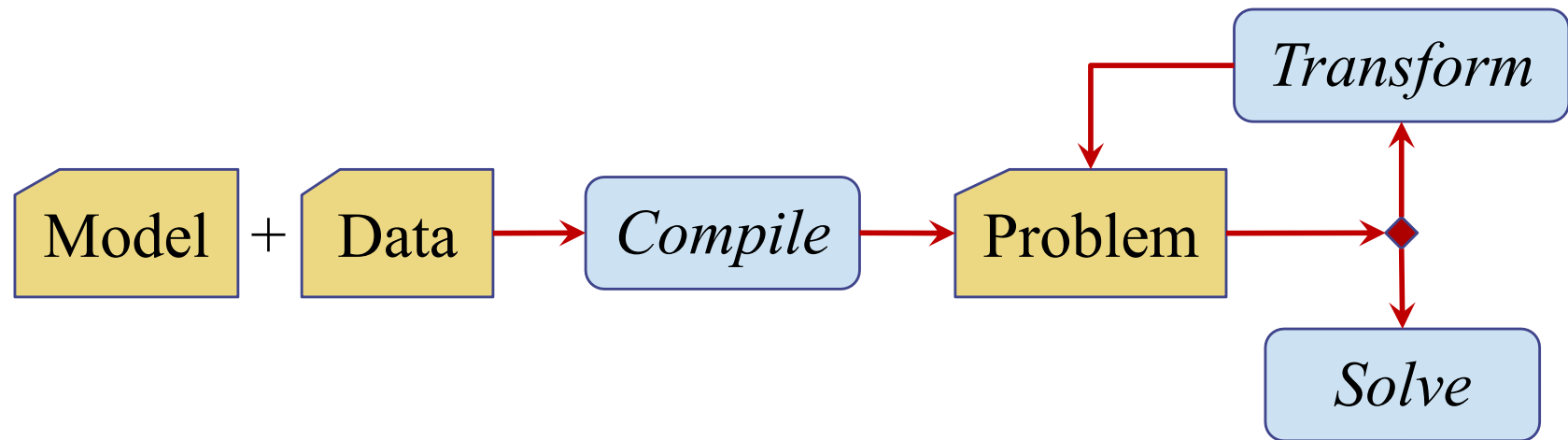
- Does any of this really encode our understanding of the *class of problems*?
  - ...or is this a reflection of our understanding of the *solver*?



# Transformations: *Projecting problems to problems*

## ■ Model Transformations

- Project from one problem space to another
- Standardize common reformulations or approximations
- Convert “unoptimizable” modeling constructs into equivalent optimizable forms



# Transformations are not entirely new

- LINGO's automatic linearization:

```
MODEL :
 MAX = @ABS(X-3);
 X <= 2;
END
```

- Generates the “usual” Big-M integer linear model:

```
MAX _C3
SUBJECT TO
 X <= 2
 - _C1 - _C2 + _C3 = 0
 _C1 - 100000 _C4 <= 0
 _C2 + 100000 _C4 <= 100000
 X - _C1 + _C2 = 3
END
INTE _C4
```

Cunningham and Schrage, “The LINGO Algebraic Modeling Language.” In *Modeling Languages in Mathematical Optimization*, Josef Kallrath ed. Springer, 2004.

# Why are we interested in transformations?

- Separate model expression from how we intend to solve it
  - Defer decisions that improve tractability until solution time
  - Explore alternative reformulations or representations
  - Support *solver-specific* model customizations (e.g., `abs()`)
  - Support iterative methods that use different solvers requiring different representations (e.g., initializing NLP from MIP)
- Support “higher level” or non-algebraic modeling constructs
  - Express models that are closer to reality, e.g.:
    - Piecewise expressions
    - Disjunctive models (switching decisions & logic models)
    - Differential-algebraic models (dynamic models)
    - Bilevel models (game theory models)
- Reduce “mechanical” errors due to manual transformation

- Disjunctions: selectively enforce sets of constraints
  - Sequencing decisions: x ends before y or y ends before x
  - Switching decisions: a process unit is built or not
  - Alternative selection: selecting from a set of pricing policies
- Implementation: leverage Pyomo “blocks”
  - **Disjunct**:
    - Block of Pyomo components
      - (Var, Param, Constraint, etc.)
    - Boolean (binary) indicator variable determines if block is enforced
  - **Disjunction**:
    - Enforces logical XOR across a set of Disjunct indicator variables
  - (Logic constraints on indicator variables)

$$\mathbf{V}_{i \in D_k} \left[ \begin{array}{c} Y_{ik} \\ h_{ik}(x) \leq o \\ c_k = \gamma_{ik} \end{array} \right] \\ \Omega(Y) = true$$

# Example: Task sequencing

- Prevent tasks colliding on a single piece of equipment
  - Derived from Raman & Grossmann (1994)
  - Given:
    - Tasks  $I$  processed on a sequence of machines (with no waiting)
    - Task  $i$  starts processing at time  $t_i$  with duration  $\tau_{im}$  on machine  $m$
    - $J(i)$  is the set of machines used by task  $i$
    - $C_{ik}$  is the set of machines used by both tasks  $i$  and  $j$

$$\left[ t_i + \sum_{\substack{m \in J(i) \\ m \leq j}} \tau_{im} \leq t_k + \sum_{\substack{m \in J(k) \\ m < j}} \tau_{km} \right] \vee \left[ t_k + \sum_{\substack{m \in J(k) \\ m \leq j}} \tau_{km} \leq t_i + \sum_{\substack{m \in J(i) \\ m < j}} \tau_{im} \right]$$

$$\forall j \in C_{ik}, \forall i, k \in I, i < k$$

# Example: Task sequencing in Pyomo

```

from pyomo.dae import *
def _NoCollision(disjunct, i, k, j, ik):
 model = disjunct.model()
 lhs = model.t[i] + sum(model.tau[i,m] for m in model.STAGES if m<j)
 rhs = model.t[k] + sum(model.tau[k,m] for m in model.STAGES if m<j)
 if ik:
 disjunct.c = Constraint(expr= lhs + model.tau[i,j] <= rhs)
 else:
 disjunct.c = Constraint(expr= rhs + model.tau[k,j] <= lhs)
model.NoCollision = Disjunction(model.L, [0,1], rule=_NoCollision)

def _setSequence(model, i, k, j):
 return [model.NoCollision[i,k,j,ik] for ik in [0,1]]
model.setSequence = Disjunction(model.L, rule=_setSequence)

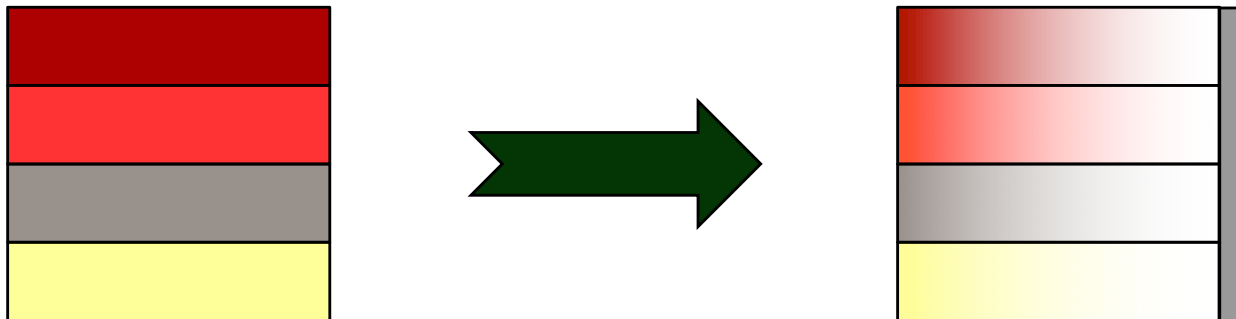
```

$$\left[ t_i + \sum_{\substack{m \in J(i) \\ m < j}} \tau_{im} + \tau_{ij} \leq t_k + \sum_{\substack{m \in J(k) \\ m < j}} \tau_{km} \right] \vee \left[ t_k + \sum_{\substack{m \in J(k) \\ m < j}} \tau_{km} + \tau_{kj} \leq t_i + \sum_{\substack{m \in J(i) \\ m < j}} \tau_{im} \right]$$

$$\forall j \in C_{ik}, \forall i, k \in I, i < k$$

# Solving disjunctive models

- Few solvers “understand” disjunctive models
  - *Transform* model into standard math program
  - Big-M relaxation:
    - Convert logic variables to binary
    - Split equality constraints in disjuncts into pairs of inequality constraints
    - Relax all constraints in the disjuncts with “appropriate” M values



```
pyomo solve --solver cbc --transform=gdp.bigm jobshop.py jobshop.dat
```

# Why is the transformation interesting?

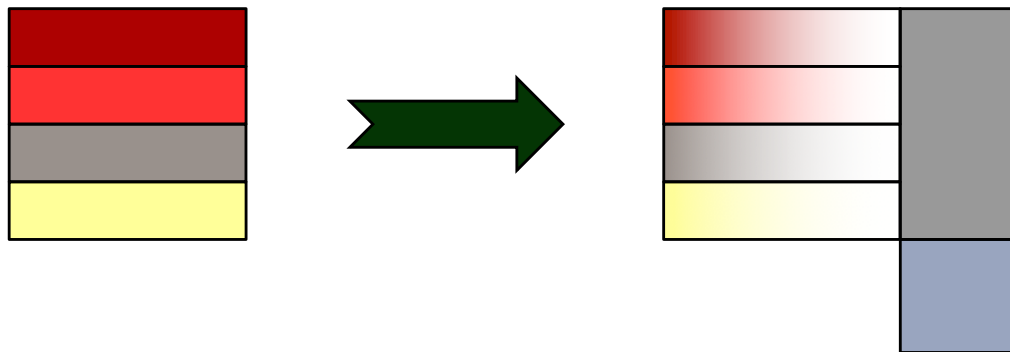
---

- Model preserves explicit disjunctive structure
- Automated transformation reduces errors
- Automatically identifies appropriate M values (for bounded linear)



# Why is the transformation interesting?

- Model preserves explicit disjunctive structure
- Automated transformation reduces errors
- Automatically identifies appropriate M values (for bounded linear)
- Big-M is not the only way to relax a disjunction!
  - Convex hull transformation (Balas, 1985; Lee and Grossmann, 2000)



```
pyomo solve --solver cbc --transform=gdp.chull jobshop.py jobshop.dat
```

- Algorithmic approaches
  - e.g., Trespalacios and Grossmann (submitted 2013)
- Prematurely choosing one relaxation makes trying others difficult

# Expression transformations: MPEC

---

- Mathematical Programming with Equilibrium Constraints (MPEC)
  - Engineering design, economic equilibrium, multilevel games
  - Feasible region may be nonconvex and disconnected
  
- Equilibrium Constraints
  - Variational inequalities
  - Complementarity conditions
  - Optimality conditions (for bilevel problems)

# MPEC formulations

- General MPEC models can be expressed as

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & f(x) \\ \text{s.t.} \quad & h(x) = 0 \\ & a_i \leq w_i(x) \leq b_i \perp v_i(x) \quad i = 1 \dots m \end{aligned}$$

- The last set of constraints are generalized mixed complementarity conditions (Ferris, Fourer, and Gay, '06), which have the form

$$\begin{aligned} & \text{either } w_i(x) = a_i \quad \text{and} \quad v_i(x) \geq 0 \\ & \quad \text{or } w_i(x) = b_i \quad \text{and} \quad v_i(x) \leq 0 \\ & \quad \text{or } a_i < w_i(x) < b_i \quad \text{and} \quad v_i(x) = 0 \end{aligned}$$

# Modeling languages support MPECs

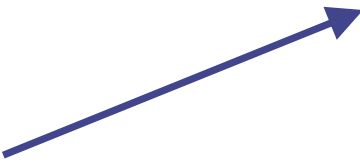
- AMPL
    - The `complements` keyword is used to denote complementarity between two constraints, expressions or variables
  - GAMS
    - The `complements` keyword is used to denote complementarity between two constraints, expressions or variables
  - AIMMS
    - Express mixed complementarity conditions by declaring complementarity variables along with associated constraints
  - YALMIP
    - The `complements` function declares a constraint that reflects a mixed complementarity condition.
- Common challenge: lack of control over how the complementarity constraints are exposed to the solver

# Complementarity conditions in Pyomo

```
from pyomo.environ import *
from pyomo.mpec import Complementarity

M = ConcreteModel()
M.x = Var(bounds=(-1,2))
M.y = Var()

M.c3 = Complementarity(expr=(M.y - M.x**2 + 1 >= 0, M.y >= 0))
```

- 
- The **Complementarity** component declares a complementarity condition
  - The tuple argument specifies the two constraints, expressions, or variables in the complementarity condition.

**This model definition is solver agnostic!**

# A simple nonlinear reformulation

---

$$\begin{array}{ll}\min & f(x) \\ \text{s.t.} & h(x) = 0 \\ & a_i \leq \omega_i \leq b_i \quad i = 1 \dots m \\ & \omega_i = w_i(x) \quad i = 1 \dots m \\ & (\omega_i - a_i)v_i(x) \leq 0 \quad i = 1 \dots m \\ & (\omega_i - b_i)v_i(x) \leq 0 \quad i = 1 \dots m\end{array}$$

- NOTE: There are serious difficulties with solving this formulation as standard stability assumptions are not met.
  - But other nonlinear transformations exist!

# A simple disjunctive reformulation

$$\begin{array}{ll}\min & f(x) \\ \text{s.t.} & h(x) = 0 \\ & \left[ \begin{array}{c} y_{1,i} \\ w_i(x) = a_i \\ v_i(x) \geq 0 \end{array} \right] \vee \left[ \begin{array}{c} y_{2,i} \\ w_i(x) = b_i \\ v_i(x) \leq 0 \end{array} \right] \vee \left[ \begin{array}{c} y_{3,i} \\ a_i < w_i(x) < b_i \\ v_i(x) = 0 \end{array} \right] \quad i = 1 \dots m \\ & y_{1,i} + y_{2,i} + y_{3,i} = 1 \quad i = 1 \dots m \\ & y_{1,i}, y_{2,i}, y_{3,i} \in \{0,1\} \quad i = 1 \dots m\end{array}$$

# Back to our original example: $\text{ABS}(x)$

- Chaining transformations

$$\begin{aligned} f = \text{abs}(x) \Rightarrow \begin{array}{l} f = x^+ + x^- \\ x = x^+ - x^- \\ x^+ \geq 0 \perp x^- \geq 0 \end{array} &\Rightarrow \left[ \begin{array}{l} Y \\ x^- = 0 \end{array} \right] \vee \left[ \begin{array}{l} \neg Y \\ x^+ = 0 \end{array} \right] \Rightarrow \begin{array}{l} f = x^+ + x^- \\ x = x^+ - x^- \\ x^- \leq My \\ x^- \leq M(1-y) \\ x^+ \geq 0, x^- \geq 0 \end{array} \end{aligned}$$

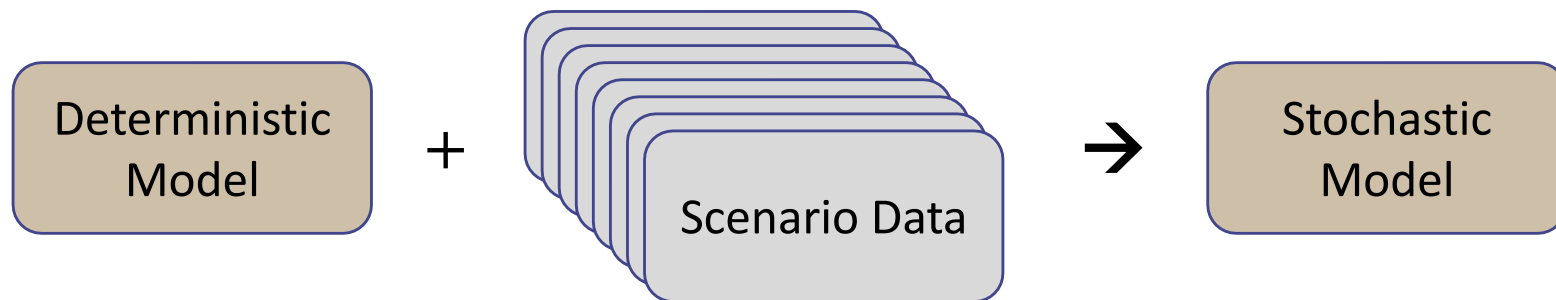
*Notional transformation chain:*

```
model = ConcreteModel()
[...]
TransformFactory("abs.complements").apply_to(model)
TransformFactory("mpec.simple_disjunction").apply_to(model)
TransformFactory("gdp.bigm").apply_to(model)
```

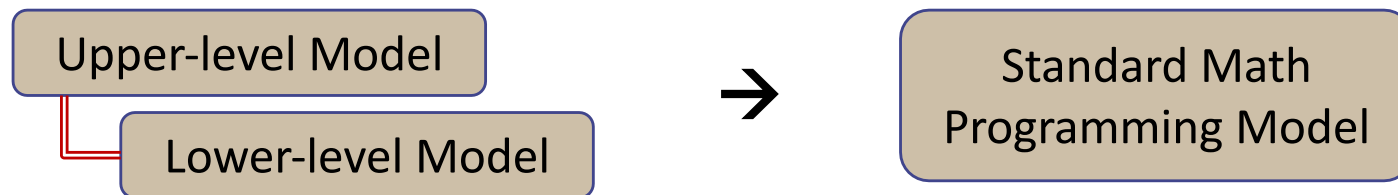


# (Pyomo) Transformations for other domains

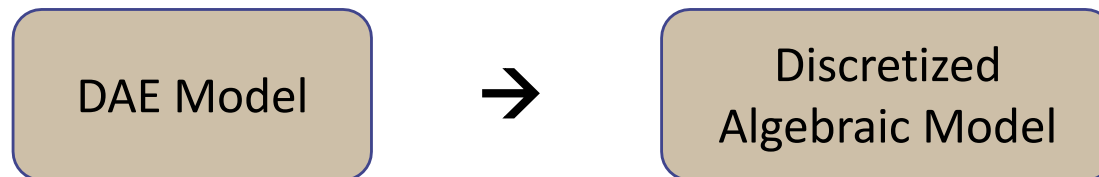
- Stochastic programming

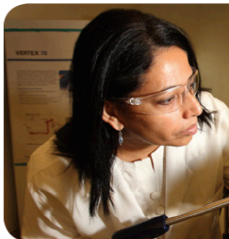
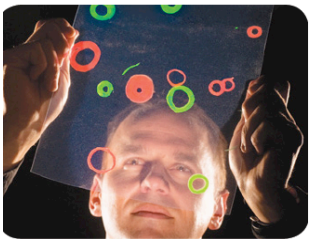


- Bilevel models



- DAE models





Sandia  
National  
Laboratories

*Exceptional  
service  
in the  
national  
interest*

# Dynamic Systems



U.S. DEPARTMENT OF  
**ENERGY**



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Examples of Dynamic Optimization

- Parameter Estimation
- Nonlinear model predictive control
- Batch process operation
- Reactor design

$$\begin{aligned} & \min \Psi(x(t), y(t), u(t)) \\ & \dot{x}(t) = f(x(t), y(t), u(t), t, p) \\ \text{DAE} \quad & 0 = g(x(t), y(t), u(t), t, p) \\ \text{model} \quad & x(t) \in \mathbb{R}^n \\ & y(t) \in \mathbb{R}^n \\ & u(t) \in \mathbb{R}^m \end{aligned}$$

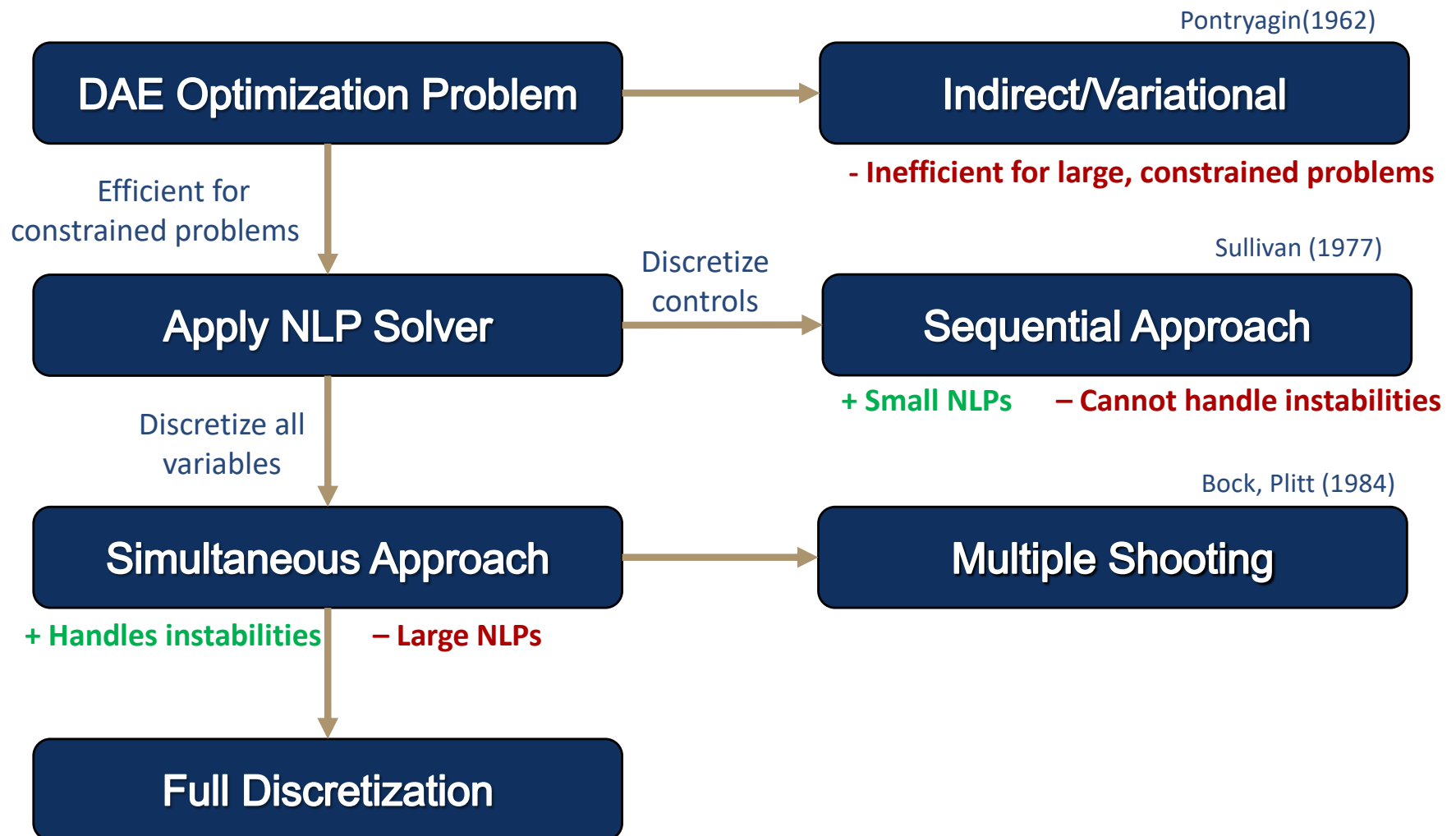
*x: State (differential) variables*

*u: Control (input) variables*

*y: Algebraic variables*

# Solution Approaches

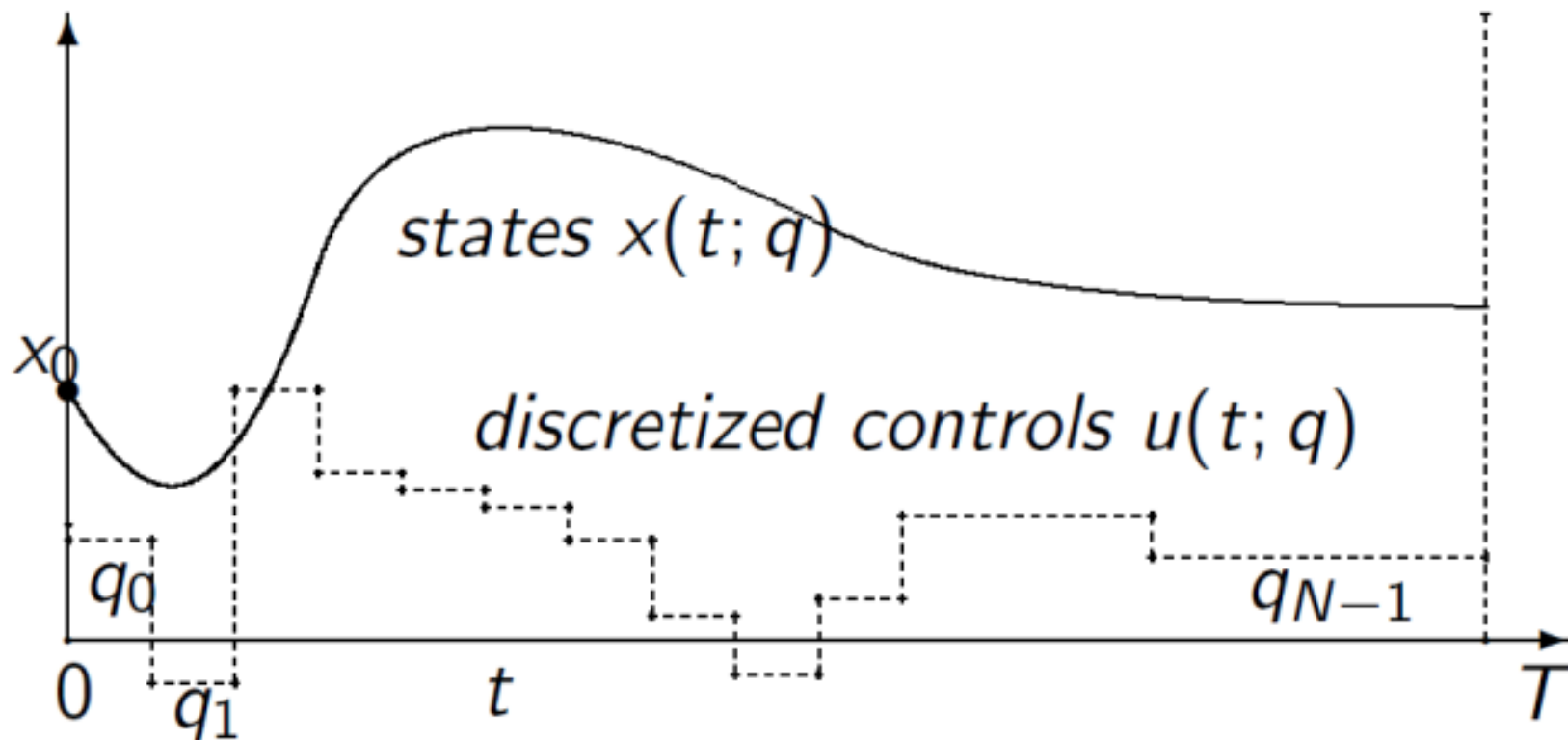
[Figure from L.T. Biegler (2007)]



# Sequential Approach

[Figure from M. Diehl]

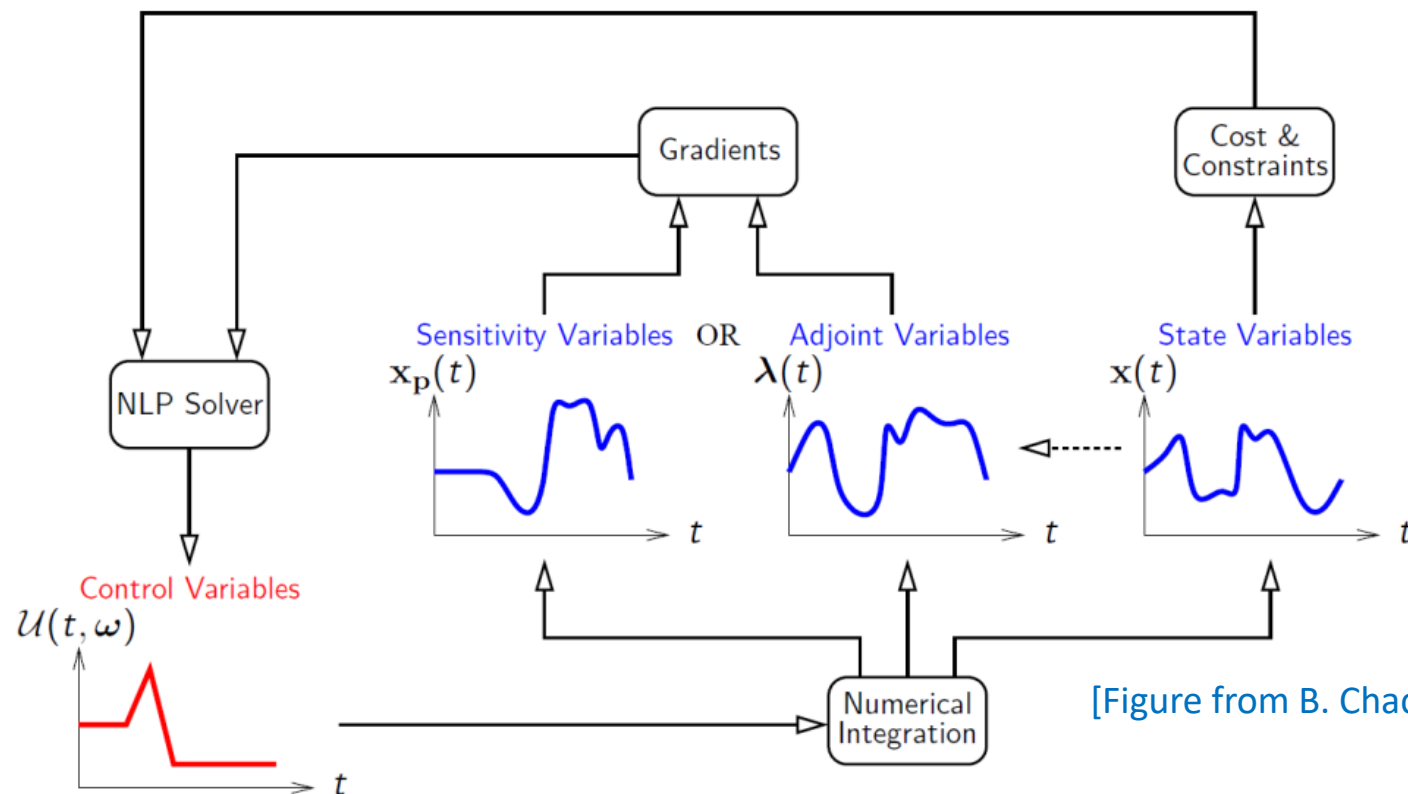
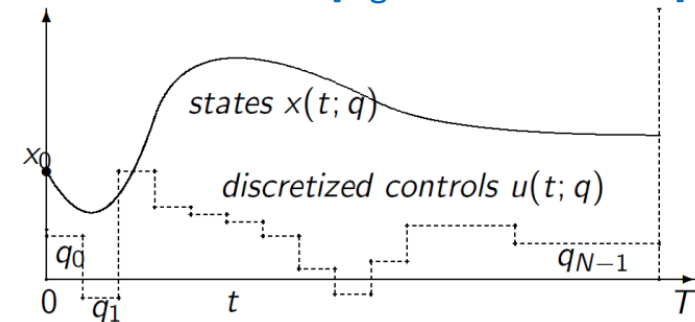
- Single Shooting Method



# Sequential Approach

- Single Shooting Method

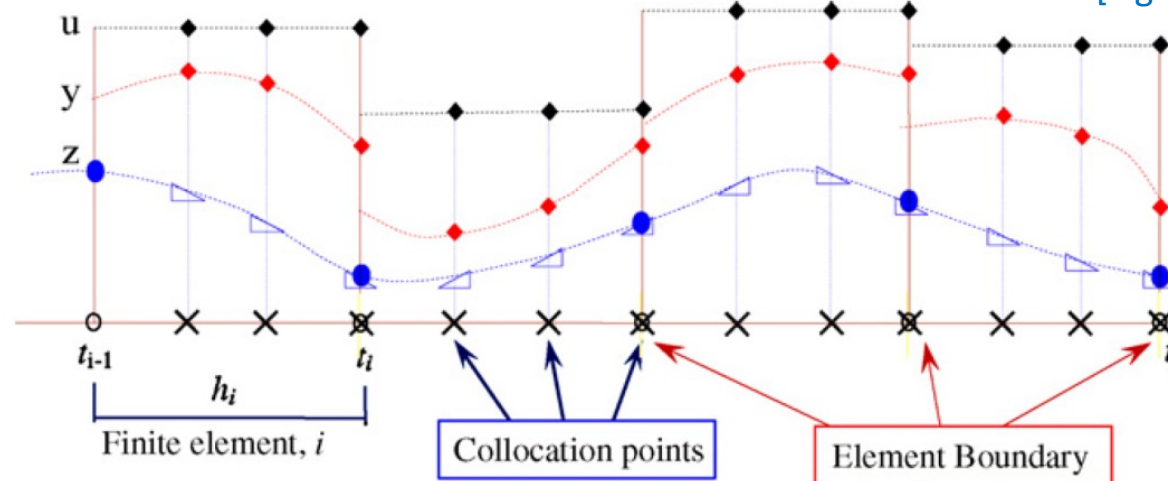
[Figure from M. Diehl]



[Figure from B. Chachuat (2009)]

# Simultaneous Approach

[Figure from L.T. Biegler (2007)]



## ■ Multiple Shooting

- Discretize controls and initial conditions for each finite element
- ✓ Embeds DAE Solvers/Sensitivity
- ✓ Can solve unstable systems
- ✓ Good for problems with long time horizons and few dynamic states
- ✗ Dense sensitivity blocks
- ✗ Difficult to enforce path constraints

## ■ Full Discretization

- Discretize all variables
- ✓ Can solve unstable systems
- ✓ Good for problems with many dynamic states and degrees of freedom
- ✓ Sparse NLP
- ✗ Large-scale NLP

# Full Discretization

- Finite Difference Methods

$$\frac{df}{dt}(t) = \lim_{h \rightarrow 0} \frac{f(t+h) - f(t)}{h}$$

Forward Difference



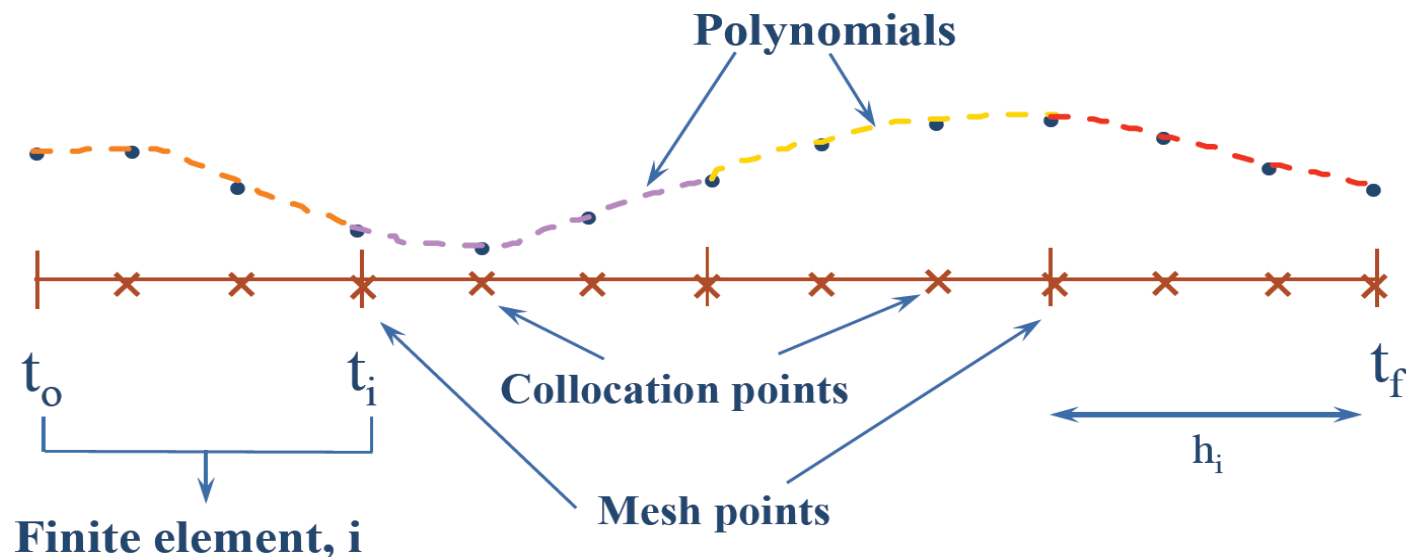
$$\frac{df}{dt}(t) = \frac{f(t+h) - f(t)}{h}$$

Backward Difference



$$\frac{df}{dt}(t) = \frac{f(t) - f(t-h)}{h}$$

- Collocation over finite elements





# Simple Example

---

$$\frac{dz}{dt} = z^2 - 2z + 1, \quad z(0) = -3$$

- **Exercise:** Solve the differential equation using a backward finite difference scheme over the time interval  $t \in [0,1]$

Backward Difference  $\frac{df}{dt}(t) = \frac{f(t) - f(t-h)}{h}$

- **Exercise:** Plot the solution against the analytic solution

Analytic solution  $z(t) = (4t - 3)/(4t + 1)$

# Simple Example – Exercise Solution

- Solve the differential equation using a backward finite difference scheme

```
from pyomo.environ import *

numpoints = 10
m = ConcreteModel()
m.points = RangeSet(0,numpoints)
m.h = Param(initialize=1.0/numpoints)

m.z = Var(m.points)
m.dzdt = Var(m.points)

m.obj = Objective(expr=1) # Dummy Objective

def _zdot(m, i):
 return m.dzdt[i] == m.z[i]**2 - 2*m.z[i] + 1
m.zdot = Constraint(m.points, rule=_zdot)

def _back_diff(m,i):
 if i == 0:
 return Constraint.Skip
 return m.dzdt[i] == (m.z[i]-m.z[i-1])/m.h
m.back_diff = Constraint(m.points, rule=_back_diff)

def _init_con(m):
 return m.z[0] == -3
m.init_con = Constraint(rule=_init_con)
```

# Simple Example – Exercise Solution

## ■ Plotting

```
solver = SolverFactory('ipopt')
solver.solve(m, tee=True)
```

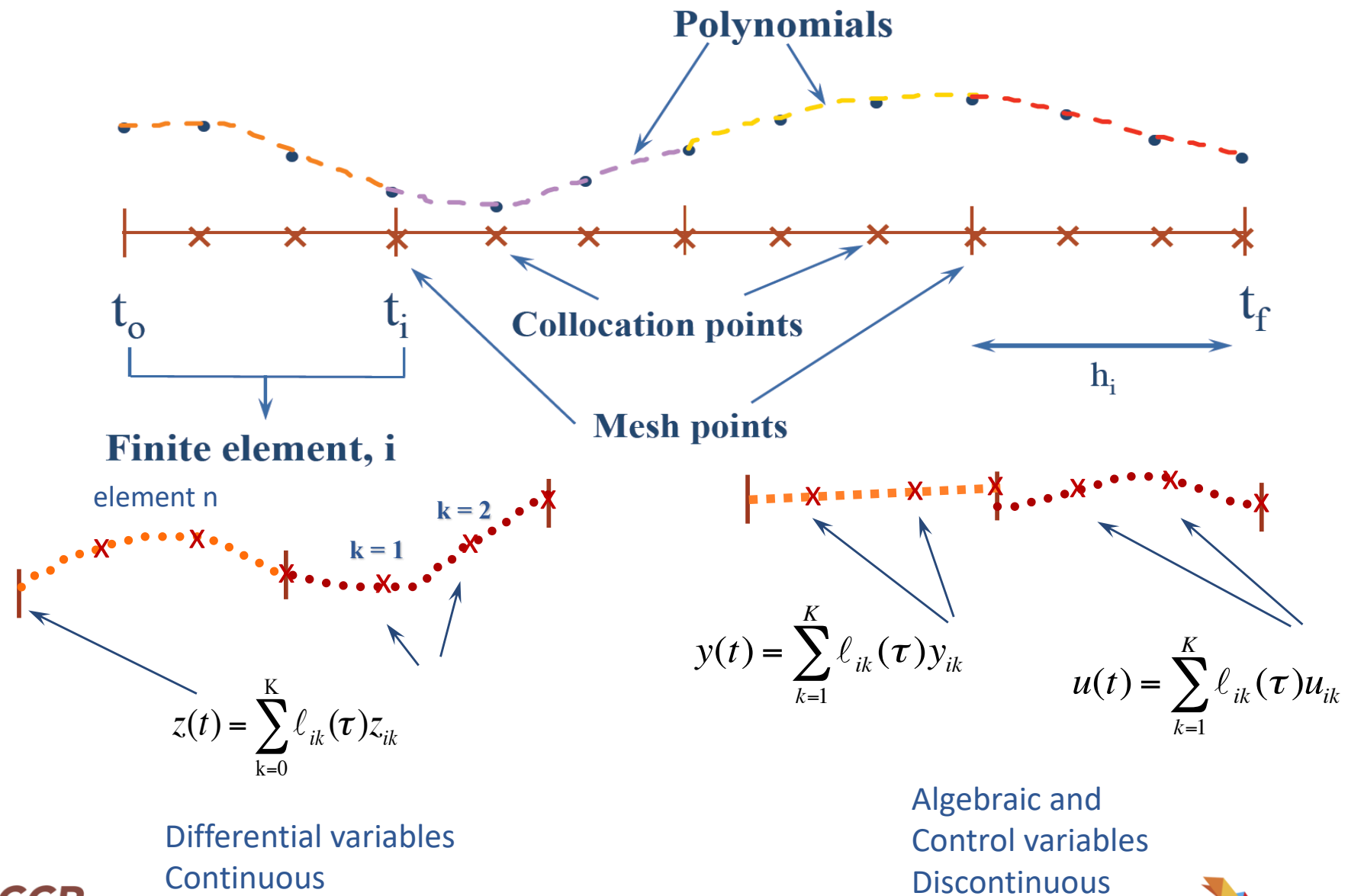
```
import matplotlib.pyplot as plt
```

```
analytical_t = [0.01*i for i in range(0,101)]
analytical_z = [(4*t-3)/(4*t+1) for t in analytical_t]
```

```
findiff_t = [m.h*i for i in m.points]
findiff_z = [value(m.z[i]) for i in m.points]
```

```
plt.plot(analytical_t, analytical_z, 'b', label='analytical solution')
plt.plot(findiff_t, findiff_z, 'ro--', label='finite difference solution')
plt.legend(loc='best')
plt.xlabel('t')
plt.show()
```

# Collocation over finite elements



# Collocation over finite elements

Given:  $\frac{dz}{dt} = f(z(t), t), \quad z(0) = z_0,$

Approximate  $z$  by Lagrange interpolation polynomials (order  $K+1$ ) with interpolation points,  $t_k$

$$\left. \begin{aligned} t &= t_{i-1} + h_i \tau, \\ z^K(t) &= \sum_{j=0}^K \ell_j(\tau) z_{ij}, \end{aligned} \right\} t \in [t_{i-1}, t_i], \quad \tau \in [0, 1], \quad \ell_j(\tau) = \prod_{k=0, k \neq j}^K \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)},$$

Collocation Equations  $\rightarrow \sum_{j=0}^K z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{ik}, t_{ik}), \quad k = 1, \dots, K,$

Collocation Coefficients to be solved for

Evaluated at  $t$  not  $\tau$

Known Collocation Points

Continuity Equations  $\rightarrow z_{i+1,0} = \sum_{j=0}^K \ell_j(1) z_{ij}, \quad i = 1, \dots, N-1$

Evaluated at the element boundary

# Collocation points

| Degree $K$ | Legendre Roots                                           | Radau Roots                                              |
|------------|----------------------------------------------------------|----------------------------------------------------------|
| 1          | 0.500000                                                 | 1.000000                                                 |
| 2          | 0.211325<br>0.788675                                     | 0.333333<br>1.000000                                     |
| 3          | 0.112702<br>0.500000<br>0.887298                         | 0.155051<br>0.644949<br>1.000000                         |
| 4          | 0.069432<br>0.330009<br>0.669991<br>0.930568             | 0.088588<br>0.409467<br>0.787659<br>1.000000             |
| 5          | 0.046910<br>0.230765<br>0.500000<br>0.769235<br>0.953090 | 0.057104<br>0.276843<br>0.583590<br>0.860240<br>1.000000 |

Shifted Gauss-Legendre and Radau roots as collocation points

# Simple Example - Collocation

$$\frac{dz}{dt} = z^2 - 2z + 1, \quad z(0) = -3$$

- Solve the differential equation using collocation with a single finite element and 3 radau collocation points over the time interval  $t \in [0,1]$

$$\sum_{j=0}^3 z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h(z_{ik}^2 - 2z_{ik} + 1), \quad k = 1, \dots, 3, \quad i = 1, \dots, N$$

$$z_{i+1,0} = \sum_{j=0}^3 \ell_j(1) z_{ij}, \quad i = 1, \dots, N-1$$

Using Radau collocation, we have  $\tau_0 = 0$ ,  $\tau_1 = 0.155051$ ,  $\tau_2 = 0.644949$  and  $\tau_3 = 1$ .

# Simple Example - Collocation

$$\sum_{j=0}^3 z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h(z_{ik}^2 - 2z_{ik} + 1), \quad k = 1, \dots, 3, \quad i = 1, \dots, N$$

$$z_{i+1,0} = \sum_{j=0}^3 \ell_j(1) z_{ij}, \quad i = 1, \dots, N - 1$$

Using Radau collocation, we have  $\tau_0 = 0$ ,  $\tau_1 = 0.155051$ ,  $\tau_2 = 0.644949$  and  $\tau_3 = 1$ .

$$\ell_0(\tau) = \frac{(\tau - \tau_1)(\tau - \tau_2)(\tau - \tau_3)}{(\tau_0 - \tau_1)(\tau_0 - \tau_2)(\tau_0 - \tau_3)} = a_3\tau^3 + a_2\tau^2 + a_1\tau + a_0$$

$$\ell_0(\tau) = -10\tau^3 + 18\tau^2 - 9\tau + 1$$

$$\dot{\ell}_0(\tau) = -30\tau^2 + 36\tau - 9$$

Other Lagrange polynomials found similarly



# Simple Example - Collocation

For  $N = 1$ , and  $z_0 = -3$  the collocation equations are given by:

$$\sum_{j=0}^3 z_j \frac{d\ell_j(\tau_k)}{d\tau} = (z_k^2 - 2z_k + 1), \quad k = 1, \dots, 3,$$

which can be written out as:

$$\begin{aligned} & z_0(-30\tau_k^2 + 36\tau_k - 9) + z_1(46.7423\tau_k^2 - 51.2592\tau_k + 10.0488) \\ & + z_2(-26.7423\tau_k^2 + 20.5925\tau_k - 1.38214) + z_3(10\tau_k^2 - \frac{16}{3}\tau_k + \frac{1}{3}) \\ & = (z_k^2 - 2z_k + 1), \quad k = 1, \dots, 3. \end{aligned}$$

Solving these three equations gives  $z_1 = -1.65701$ ,  $z_2 = 0.032053$ ,  $z_3 = 0.207272$

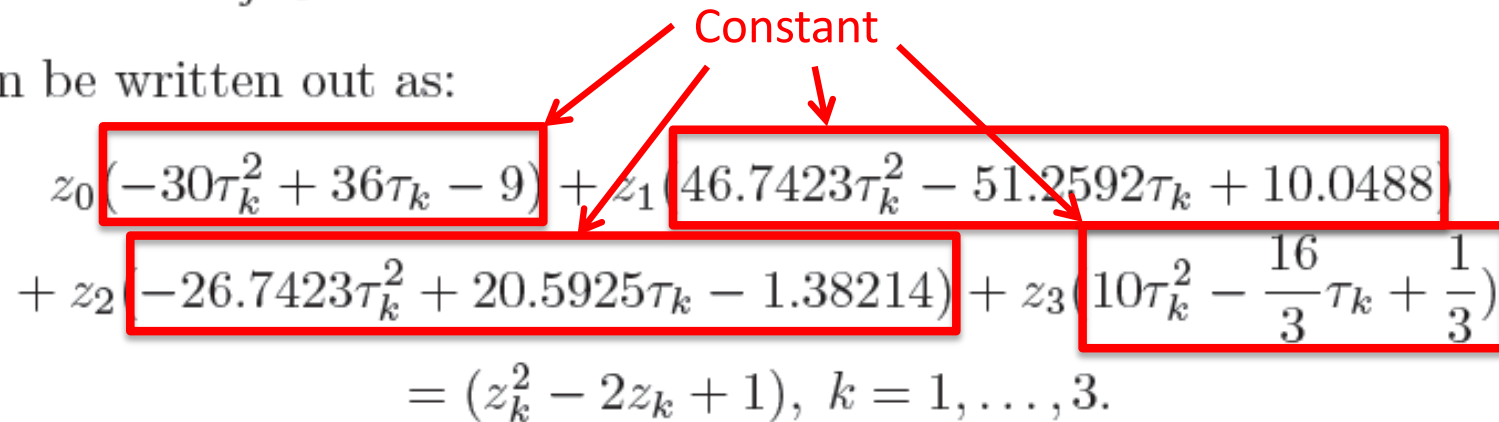
# Collocation Matrix

$$\sum_{j=0}^3 z_j \frac{d\ell_j(\tau_k)}{d\tau} = (z_k^2 - 2z_k + 1), \quad k = 1, \dots, 3,$$

which can be written out as:

$$\begin{aligned}
 & z_0(-30\tau_k^2 + 36\tau_k - 9) + z_1(46.7423\tau_k^2 - 51.2592\tau_k + 10.0488) \\
 & + z_2(-26.7423\tau_k^2 + 20.5925\tau_k - 1.38214) + z_3(10\tau_k^2 - \frac{16}{3}\tau_k + \frac{1}{3}) \\
 & = (z_k^2 - 2z_k + 1), \quad k = 1, \dots, 3.
 \end{aligned}$$

Constant



$$\blacksquare \quad \dot{a}(k, k) = \begin{bmatrix} \dot{\ell}_0(\tau_0) & \cdots & \dot{\ell}_0(\tau_k) \\ \vdots & \ddots & \vdots \\ \dot{\ell}_k(\tau_0) & \cdots & \dot{\ell}_k(\tau_k) \end{bmatrix}$$

# Python code for generating collocation matrix

```
import numpy
Specify collocation points
cp = [0, 0.155051, 0.644949, 1]

a = []
for i in range(len(cp)):
 ptmp = []
 tmp = 0
 for j in range(len(cp)):
 if j != i:
 row = []
 row.insert(0, 1/(cp[i]-cp[j]))
 row.insert(1, -cp[j]/(cp[i]-cp[j]))
 ptmp.insert(tmp, row)
 tmp += 1
 p=[1]
 for j in range(len(cp)-1):
 p = numpy.convolve(p,ptmp[j])
 pder = numpy.polyder(p,1)
 arow = []
 for j in range(len(cp)):
 arow.append(numpy.polyval(pder,cp[j]))
 a.append(arow)
print(arow)
```

# Simple Example - Collocation

$$\frac{dz}{dt} = z^2 - 2z + 1, \quad z(0) = -3$$

- **Exercise:** Solve the differential equation using collocation over a single finite element with  $t \in [0,1]$

$$\left. \begin{aligned} t &= t_{i-1} + h_i \tau, \\ z^K(t) &= \sum_{j=0}^K \ell_j(\tau) z_{ij}, \end{aligned} \right\} t \in [t_{i-1}, t_i], \quad \tau \in [0, 1], \quad \ell_j(\tau) = \prod_{k=0, k \neq j}^K \frac{(\tau - \tau_k)}{(\tau_j - \tau_k)},$$

Collocation Equations  $\rightarrow \sum_{j=0}^K z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{ik}, t_{ik}), \quad k = 1, \dots, K,$

- **Exercise:** Plot the solution against the analytic solution

Analytic solution  $z(t) = (4t - 3)/(4t + 1)$

# Implementation is challenging!

---

- Common theme: significant effort to rework formulation
  - Time: first ~6 months of a grad student's research
  - Error prone: many ways to make subtle mistakes
  - Inflexible: formulation specific to selected solution approach
- Difficult to know apriori the best solution approach for a particular model

# Expressing dynamical systems

[with J. Sirola, V. Zavala]

- Model dynamical systems in a natural form
  - Systems of Differential Algebraic Equations (DAE)
  - Extend the Pyomo component model
    - ContinuousSet: A virtual set over which you can take a derivative
    - DerivativeVar: The derivative of a Var with respect to a ContinuousSet

$$\begin{array}{ll} \min & \Psi(x(t), y(t), u(t)) \\ & \dot{x}(t) = f(x(t), y(t), u(t), t, p) \\ \text{DAE} & 0 = g(x(t), y(t), u(t), t, p) \\ \text{model} & x(t) \in \mathcal{R}^n \\ & y(t) \in \mathcal{R}^n \\ & u(t) \in \mathcal{R}^m \end{array}$$

# Simple Example – pyomo.dae

```
from pyomo.environ import *
from pyomo.dae import *

model = m = ConcreteModel()
m.t = ContinuousSet(bounds=(0, 1))

m.z = Var(m.t)
m.dzdt = DerivativeVar(m.z, wrt=m.t)

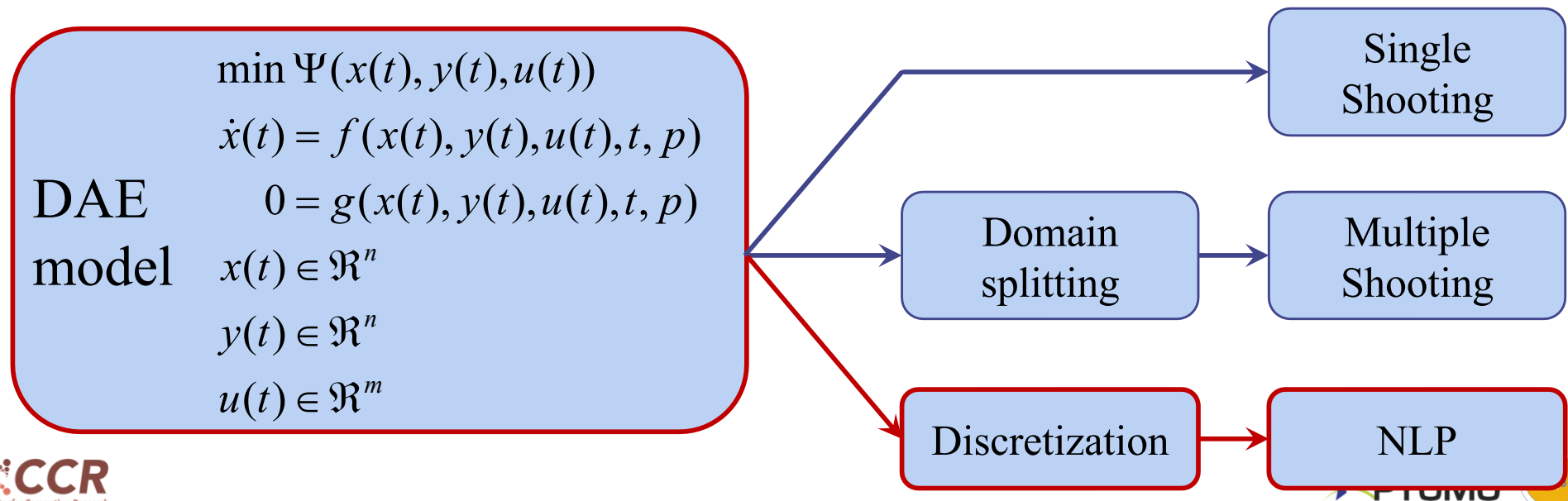
m.obj = Objective(expr=1) # Dummy Objective

def _zdot(m, t):
 if t == 0:
 return Constraint.Skip
 return m.dzdt[t] == m.z[t]**2 - 2*m.z[t] + 1
m.zdot = Constraint(m.t, rule=_zdot)

def _init_con(m):
 return m.z[0] == -3
m.init_con = Constraint(rule=_init_con)
```

# Solving dynamical systems

- Given that we have a DAE model in Pyomo... now what?
  - How to optimize?
    - Simulation-based / Multiple shooting methods / Simultaneous
  - Common theme: significant effort to rework formulation
    - Time / Error prone / Inflexible
- Our approach: separate the *declaration* of dynamical models from the *solution approach* using (nearly) *automatic transformations*





# Simple Example – pyomo.dae

```
from pyomo.environ import *
from pyomo.dae import *

model = m = ConcreteModel()
m.t = ContinuousSet(bounds=(0, 1))

m.z = Var(m.t)
m.dzdt = DerivativeVar(m.z, wrt=m.t)

m.obj = Objective(expr=1) # Dummy Objective

def _zdot(m, t):
 if t == 0:
 return Constraint.Skip
 return m.dzdt[t] == m.z[t]**2 - 2*m.z[t] + 1
m.zdot = Constraint(m.t, rule=_zdot)

def _init_con(m):
 return m.z[0] == -3
m.init_con = Constraint(rule=_init_con)

Discretize model using backward finite difference
discretizer = TransformationFactory('dae.finite_difference')
discretizer.apply_to(m, nfe=10, scheme='BACKWARD')

Discretize model using radau collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m, nfe=1, ncp=3, scheme='LAGRANGE-RADAU')
```

# Small example: optimal control

(1/8)



$$\begin{aligned} \min \quad & x_3(t_f) \\ \text{s.t.} \quad & \dot{x}_1 = x_2 \\ & \dot{x}_2 = -x_2 + u \\ & \dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2 \\ & x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0 \\ & x_1(0) = 0 \\ & x_2(0) = -1 \\ & x_3(0) = 0 \\ & t_f = 1 \end{aligned}$$

[Jacobson and Lele (1969)]

```
from pyomo.environ import *
from pyomo.dae import *

model = m = ConcreteModel()
m.tf = Param(initialize = 1)
m.t = ContinuousSet(bounds=(0, m.tf))

m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

m.obj = Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
m.x1dot = Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
 return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)

def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

# Small example: optimal control

(2/8)

$$\begin{aligned} \min \quad & x_3(t_f) \\ \text{s.t.} \quad & \dot{x}_1 = x_2 \\ & \dot{x}_2 = -x_2 + u \\ & \dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2 \\ & x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0 \\ & x_1(0) = 0 \\ & x_2(0) = -1 \\ & x_3(0) = 0 \\ & t_f = 1 \end{aligned}$$

[Jacobson and Lele (1969)]

```
from pyomo.environ import *
from pyomo.dae import *

model = m = ConcreteModel()
m.tf = Param(initialize = 1)
m.t = ContinuousSet(bounds=(0, m.tf))

m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
```

```
from pyomo.environ import *
from pyomo.dae import *
```

```
m.obj = Objective(expr=m.x3[m.tf])

def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
m.x1dot = Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
 return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)

def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

# Small example: optimal control

(3/8)

```
from pyomo.environ import *
from pyomo.dae import *
```

```
model = m = ConcreteModel()
m.tf = Param(initialize = 1)
m.t = ContinuousSet(bounds=(0, m.tf))
```

```
m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
```

```
n
s.t
x2
model = m = ConcreteModel()
m.tf = Param(initialize = 1)
m.t = ContinuousSet(bounds=(0, m.tf))
```

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

```
def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
m.x1dot = Constraint(m.t, rule=_x1dot)

def _x2dot(m, t):
 return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = Constraint(m.t, rule=_x2dot)

def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)

def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)

def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

# Small example: optimal control

(4/8)

$$\begin{aligned}
 &\min x_3(t_f) \\
 &s.t. \quad \dot{x}_1 = x_2 \\
 &\quad \dot{x}_2 = -x_2 + u \\
 &\quad \dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2 \\
 &x_2 - 8 * (t - t_f) \\
 &x_1(0) \\
 &x_2(0) \\
 &x_3(0)
 \end{aligned}$$

[Jacobson and

```

from pyomo.environ import *
from pyomo.dae import *

model = m = ConcreteModel()
m.tf = Param(initialize=1)
m.t = ContinuousSet(bounds=(0, m.tf))

```

```

m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

```

```

m.obj = Objective(expr=m.x3[m.tf])

```

```

def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]

```

```

m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)

```

```

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)

```

```

yield m.x2[0] == -1
yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)

```

# Small example: optimal control

(5/8)

$$\min x_3(t_f)$$

$$s.t. \quad \dot{x}_1 = x_2$$

$$\dot{x}_2 = -x_2 + u$$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2$$

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

```
from pyomo.environ import *
from pyomo.dae import *

model = m = ConcreteModel()
m.tf = Param(initialize = 1)
m.t = ContinuousSet(bounds=(0, m.tf))

m.u = Var(m.t, initialize=0)
m.x1 = Var(m.t)
m.x2 = Var(m.t)
m.x3 = Var(m.t)

m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)
```

```
m.obj = Objective(expr=m.x3[m.tf])
```

```
def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
```

```
m.obj = Objective(expr=m.x3[m.tf])
```

```
def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)
```

```
def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

```
def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

# Small example: optimal control

(6/8)

from pyomo.environ import \*

```
def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)
```

min  $x_3(t_f)$

s.t.  $\dot{x}_1 = x_2$

$\dot{x}_2 = -x_2 + u$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

```
m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)
```

```
m.obj = Objective(expr=m.x3[m.tf])
```

```
def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
m.x1dot = Constraint(m.t, rule=_x1dot)
```

```
def _x2dot(m, t):
 return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = Constraint(m.t, rule=_x2dot)
```

```
def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)
```

```
def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

```
def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

# Small example: optimal control

(7/8)

```
def _con(m,t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

$$s.t. \quad \dot{x}_1 = x_2$$

$$\dot{x}_2 = -x_2 + u$$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

$$x_1(0) = 0$$

$$x_2(0) = -1$$

$$x_3(0) = 0$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

```
from pyomo.environ import *
from pyomo.dae import *
```

```
model = m = ConcreteModel()
```

```
m.dx1dt = DerivativeVar(m.x1, wrt=m.t)
m.dx2dt = DerivativeVar(m.x2, wrt=m.t)
m.dx3dt = DerivativeVar(m.x3, wrt=m.t)
```

```
m.obj = Objective(expr=m.x3[m.tf])
```

```
def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
m.x1dot = Constraint(m.t, rule=_x1dot)
```

```
def _x2dot(m, t):
 return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = Constraint(m.t, rule=_x2dot)
```

```
def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)
```

```
def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

```
def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```



# Small example: optimal control

(8/8)

```
from pyomo.environ import *
from pyomo.dae import *
```

```
model = m = ConcreteModel()
```

```
def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

$$\dot{x}_2 = -x_2 + u$$

$$\dot{x}_3 = x_1^2 + x_2^2 + 0.005 * u^2$$

$$x_2 - 8 * (t - 0.5)^2 + 0.5 \leq 0$$

$$\begin{aligned} x_1(0) &= 0 \\ x_2(0) &= -1 \\ x_3(0) &= 0 \end{aligned}$$

$$t_f = 1$$

[Jacobson and Lele (1969)]

```
m.obj = Objective(expr=m.x3[m.tf])
```

```
def _x1dot(m, t):
 return m.dx1dt[t] == m.x2[t]
m.x1dot = Constraint(m.t, rule=_x1dot)
```

```
def _x2dot(m, t):
 return m.dx2dt[t] == -m.x2[t] + m.u[t]
m.x2dot = Constraint(m.t, rule=_x2dot)
```

```
def _x3dot(m, t):
 return m.dx3dt[t] == m.x1[t]**2 + \
 m.x2[t]**2 + 0.005*m.u[t]**2
m.x3dot = Constraint(m.t, rule=_x3dot)
```

```
def _con(m, t):
 return m.x2[t] - 8*(t-0.5)**2 + 0.5 <= 0
m.con = Constraint(m.t, rule=_con)
```

```
def _init(m):
 yield m.x1[0] == 0
 yield m.x2[0] == -1
 yield m.x3[0] == 0
m.init_conditions = ConstraintList(rule=_init)
```

# Optimal Control Example - Script

```
from pyomo.environ import *
Import dynamic model
from optimalControl import m

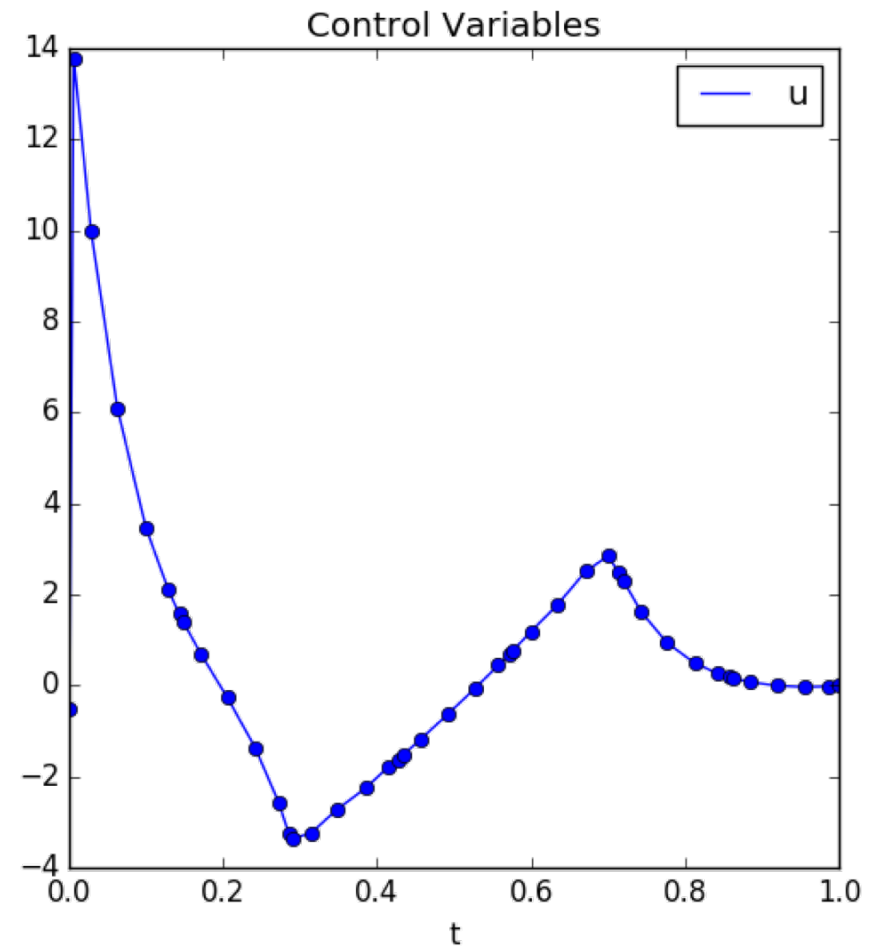
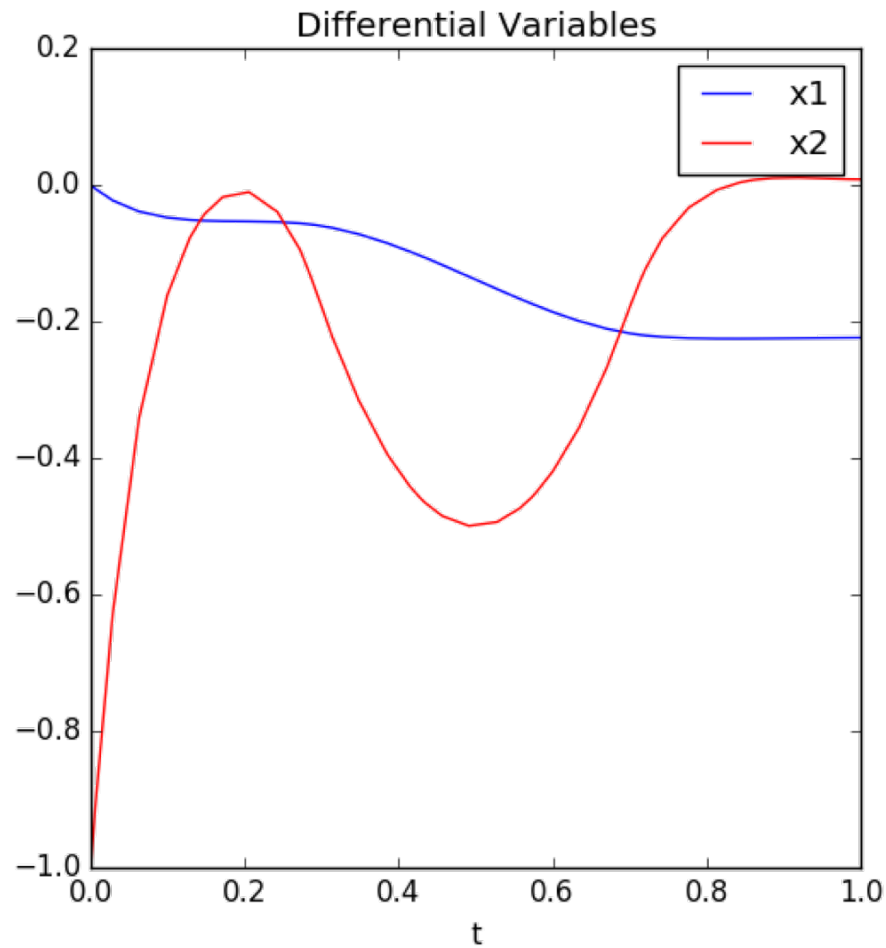
Discretize model using radau collocation
TransformationFactory('dae.collocation').apply_to(
 m, nfe=7, ncp=6, scheme='LAGRANGE-RADAU')

Solve algebraic model
results = SolverFactory('ipopt').solve(m)

def plotter(subplot, x, *series, **kws):
 plt.subplot(subplot)
 for i,y in enumerate(series):
 plt.plot(x, [value(y[t]) for t in x], 'b' + kws.get('points', ''))
 plt.title(kws.get('title', ''))
 plt.legend(tuple(y.cname() for y in series))
 plt.xlabel(x.cname())

import matplotlib.pyplot as plt
plotter(121, m.t, m.x1, m.x2, title='Differential Variables')
plotter(122, m.t, m.u, title='Control Variable', points='o')
plt.show()
```

# Optimal Control Example - Results



# Optimal Control Example - Script

```
from pyomo.environ import *
Import dynamic model
from optimalControl import m

Discretize model using radau collocation
discretizer = TransformationFactory('dae.collocation')
discretizer.apply_to(m, nfe=7, ncp=6, scheme='LAGRANGE-RADAU')

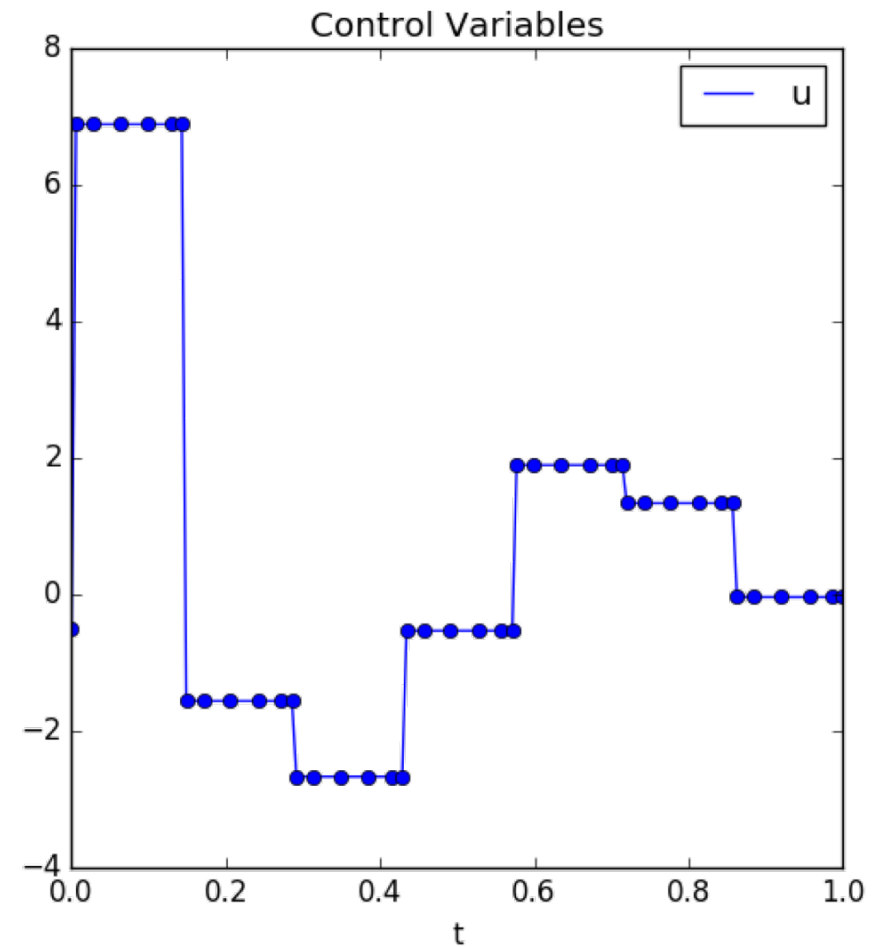
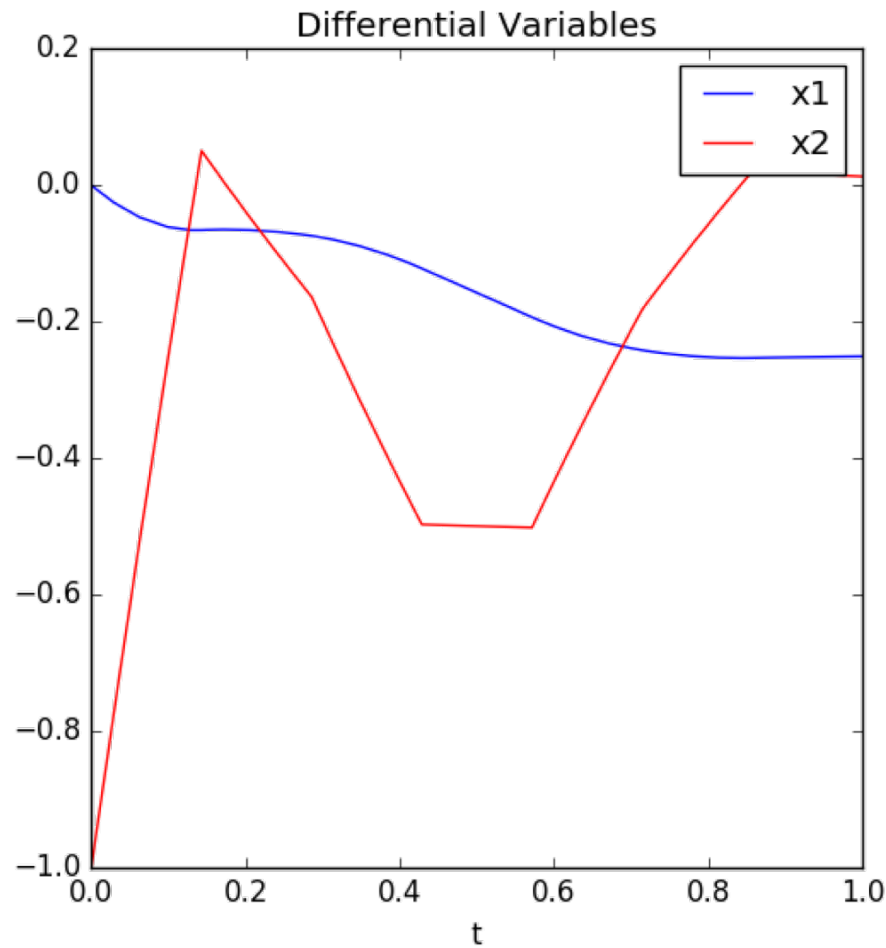
Control variable u made constant over each finite element
discretizer.reduce_collocation_points(var=m.u, ncp=1, contset=m.t)

Solve algebraic model
results = SolverFactory('ipopt').solve(m)

def plotter(subplot, x, *series, **kws):
 plt.subplot(subplot)
 for i,y in enumerate(series):
 plt.plot(x, [value(y[t]) for t in x], 'b' + kws.get('points', ''))
 plt.title(kws.get('title', ''))
 plt.legend(tuple(y.cname() for y in series))
 plt.xlabel(x.cname())

import matplotlib.pyplot as plt
plotter(121, m.t, m.x1, m.x2, title='Differential Variables')
plotter(122, m.t, m.u, title='Control Variable', points='o')
plt.show()
```

# Optimal Control Example - Results 2



# Parameter Estimation Example 1

$$\min \sum_{t_i} \left( x_1(t_i) - x_{1_{meas}}(t_i) \right)^2$$

$$s. t. \quad \frac{dx_1}{dt} = x_2$$

$$\frac{dx_2}{dt} = 1 - 2x_2 - x_1$$

$$-1.5 \leq p_1, p_2 \leq 1.5$$

$$x_1(0) = p_1, \quad x_2(0) = p_2$$

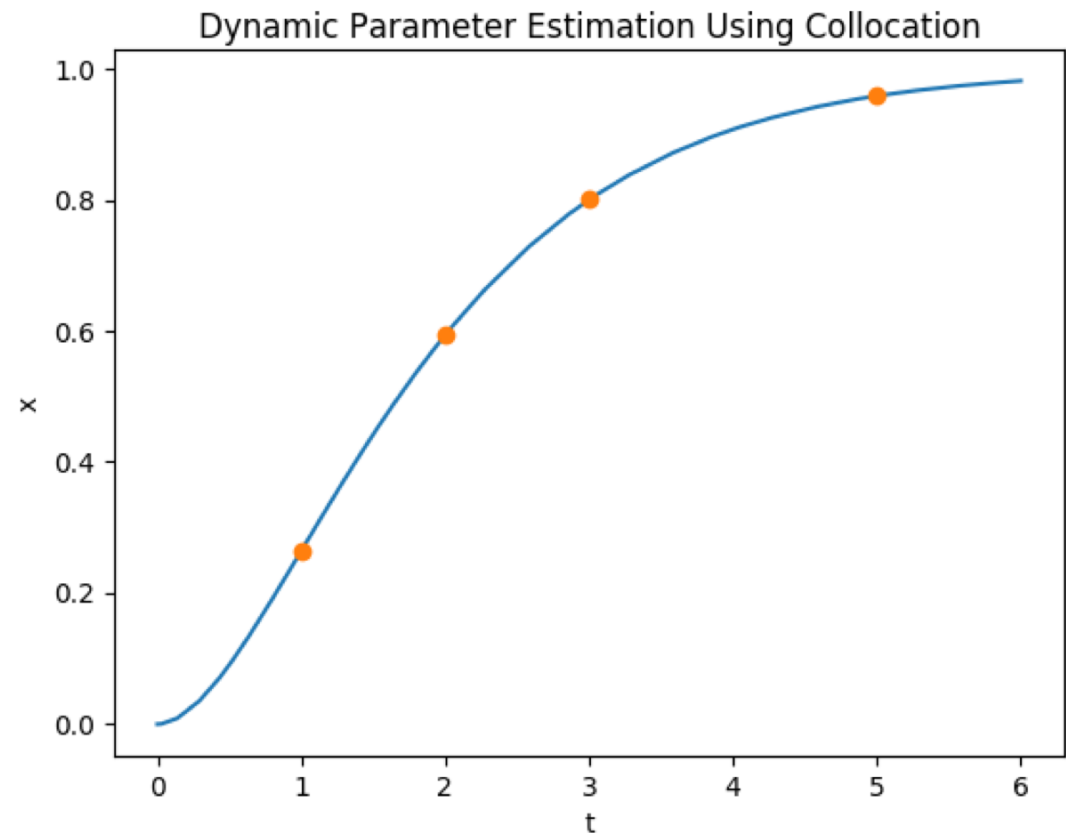
| Time           | 1     | 2     | 3     | 5     |
|----------------|-------|-------|-------|-------|
| $x_{1_{meas}}$ | 0.264 | 0.594 | 0.801 | 0.959 |

# Parameter Estimation Example 1

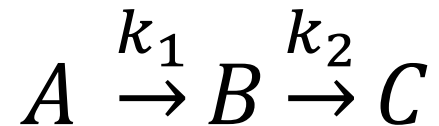
## data file

```
set MEAS_t := 1 2 3 5 ;
param x1_meas :=
1 0.264
2 0.594
3 0.801
5 0.959
;

param tlb := 0;
param tub := 6;
```



# Parameter Estimation Example 2



$$\frac{dA}{dt} = -k_1 A$$

$$\frac{dB}{dt} = k_1 A - k_2 B$$

$$A(0) = 1, \quad B(0) = 0$$

| Time | 0.1   | 0.2   | 0.3   | 0.4   | 0.5   | 0.6   | 0.7   | 0.8   | 0.9   | 1.0   |
|------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| A    | 0.606 | 0.368 | 0.223 | 0.135 | 0.082 | 0.050 | 0.030 | 0.018 | 0.011 | 0.007 |
| B    | 0.373 | 0.564 | 0.647 | 0.669 | 0.656 | 0.624 | 0.583 | 0.539 | 0.494 | 0.451 |

- **Exercise:** Solve for  $k_1$  and  $k_2$  given the concentration measurements in the table

**Solution to this problem is  $k_1 = 5$ ,  $k_2 = 1$**



# Disease Transmission Example

- Parameter estimation of a S-I-R disease transmission model<sup>[\*]</sup>

$$\begin{aligned}
 & \min \quad \omega_M \sum_{i \in \mathcal{F}} (\ln(\varepsilon_{M_i}))^2 + \omega_Q \sum_{k \in \mathcal{T}} (\varepsilon_{Q_k})^2 \\
 & \text{s.t.} \quad \frac{dS}{dt} = \frac{-\beta(y(t))S(t)I(t)}{N(t)} \cdot \varepsilon_M(t) + B(t), \\
 & \quad \frac{dI}{dt} = \frac{\beta(y(t))S(t)I(t)}{N(t)} \cdot \varepsilon_M(t) - \gamma I(t), \\
 & \quad \frac{dQ}{dt} = \frac{\beta(y(t))S(t)I(t)}{N(t)} \cdot \varepsilon_M(t), \\
 & \quad R_k^\star = \eta_k(Q_{i,k} - Q_{i,k-1}) + \varepsilon_{Q_k}, \\
 & \quad \bar{S} = \frac{\sum_{i \in \mathcal{F}} S_i}{\text{len}(\mathcal{F})}, \\
 & \quad \bar{\beta} = \frac{\sum_{i \in \tau} \beta_i}{\text{len}(\tau)}, \\
 & \quad 0 \leq I(t), S(t) \leq N(t) \\
 & \text{and} \quad 0 \leq \beta(y(t)), Q(t),
 \end{aligned}$$

Discretized problem:

- 3 differential equations
- 520 finite elements
- 3 collocation points
- ~ 10,500 variables
- ~ 10,000 constraints

# Performance impact

- Comparing the automated discretization to a manually discretized model implemented (and tuned) by a person

|                                   | Manual<br>Discretization | Using pyomo.dae<br>(Radau Collocation) |
|-----------------------------------|--------------------------|----------------------------------------|
| Model Creation Time<br>(CPU secs) | 1.79                     | 5.29                                   |
| Solve Time<br>(CPU secs)          | 1.35                     | 0.86                                   |
| IPOPT Iterations                  | 27                       | 26                                     |
| Objective ( $\times 10^{-5}$ )    | 1.4716                   | 1.4716                                 |

- The bulk of the added model creation time is the model transformation that implements the discretization

# Distillation Example

```

model.S_TRAYS = Set()
model.S_RECTIFICATION = Set(within = model.S_TRAYS)
model.S_STRIPPING = Set(within = model.S_TRAYS)

model.t = ContinuousSet(initialize=range(1,52))
model.x = Var(model.S_TRAYS, model.t, initialize=x_init_rule)
model.dx = DerivativeVar(model.x)

def _diffeq(m,n,t):
 if t == 1:
 return Constraint.Skip
 if n == 1:
 return m.dx[n,t] == 1/m.acond*m.V[t]*(m.y[n+1,t]-m.x[n,t])
 elif n in m.S_RECTIFICATION:
 return m.dx[n,t] == 1/m.atray*(m.L[t]*(m.x[n-1,t]-m.x[n,t])-m.V[t]*(m.y[n,t]-m.y[n+1,t]))
 elif n == 17:
 return m.dx[n,t] == 1/m.atray*(m.Feed*m.x_Feed+m.L[t]*m.x[n-1,t]-m.FL[t]*m.x[n,t]- \
 m.V[t]*(m.y[n,t]-m.y[n+1,t]))
 elif n in m.S_STRIPPING:
 return m.dx[n,t] == 1/m.atray*(m.FL[t]*(m.x[n-1,t]-m.x[n,t])-m.V[t]*(m.y[n,t]-m.y[n+1,t]))
 else :
 return m.dx[n,t] == 1/m.areb*(m.FL[t]*m.x[n-1,t]-(m.Feed-m.D)*m.x[n,t]-m.V[t]*m.y[n,t])
model.diffeq = Constraint(model.S_TRAYS, model.t, rule=_diffeq)

```

# PDE Example

- Illustrative example<sup>[1]</sup>

- PDE

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

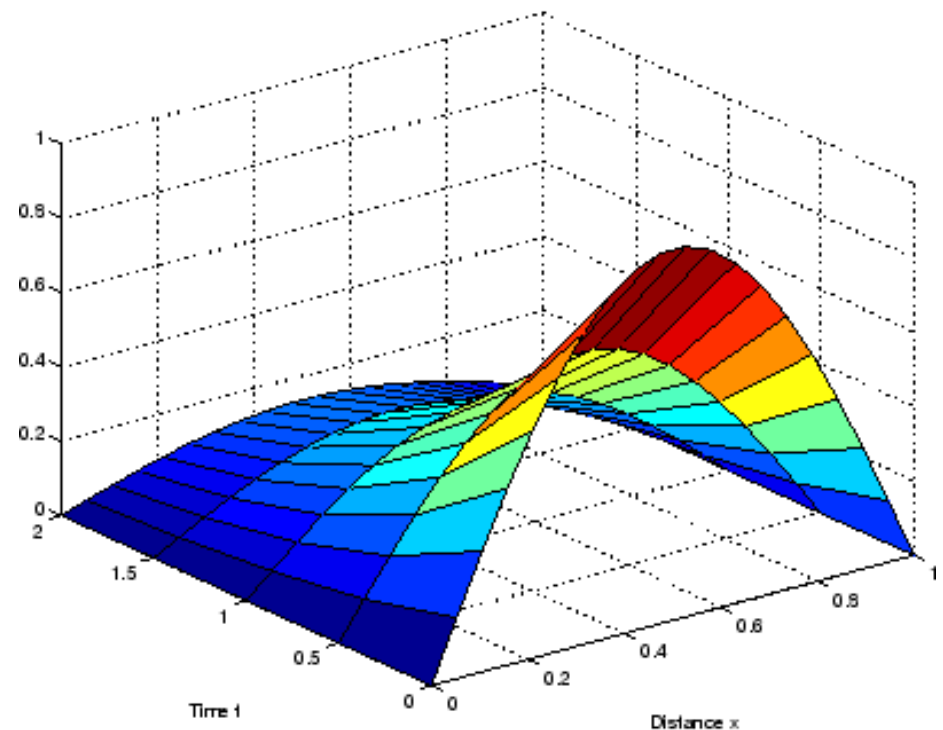
- Initial Condition

$$u(x, 0) = \sin(\pi x)$$

- Boundary Conditions

$$u(0, t) = 0$$

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$



# PDE Example

```

from pyomo.environ import *
from pyomo.dae import *

m = ConcreteModel()
m.pi = Param(initialize=3.1416)
m.t = ContinuousSet(bounds=(0,2))
m.x = ContinuousSet(bounds=(0,1))
m.u = Var(m.x,m.t)

Declare derivatives in the model
m.dudx = DerivativeVar(m.u,wrt=m.x)
m.dudx2 = DerivativeVar(m.u,wrt=(m.x,m.x))
m.dudt = DerivativeVar(m.u,wrt=m.t)

```

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad u(x, 0) = \sin(\pi x) \quad u(0, t) = 0 \quad \pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0$$

# PDE Example

$$\pi^2 \frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left( \frac{\partial u}{\partial x} \right) \rightarrow$$

```
def _pde(m,i,j):
 if i == 0 or i == 1 or j == 0 :
 return Constraint.Skip
 return m.pi**2*m.dudt[i,j] == m.dudx2[i,j]
m.pde = Constraint(m.x,m.t,rule=_pde)
```

$$u(x, 0) = \sin(\pi x) \rightarrow$$

```
def _initcon(m,i):
 if i == 0 or i == 1:
 return Constraint.Skip
 return m.u[i,0] == sin(m.pi*i)
m.initcon = Constraint(m.x,rule=_initcon)
```

$$u(0, t) = 0 \rightarrow$$

```
def _lowerbound(m,j):
 return m.u[0,j] == 0
m.lowerbound = Constraint(m.t,rule=_lowerbound)
```

$$\pi e^{-t} + \frac{\partial u}{\partial x}(1, t) = 0 \rightarrow$$

```
def _upperbound(m,j):
 return m.pi*exp(-j)+m.dudx[1,j] == 0
m.upperbound = Constraint(m.t,rule=_upperbound)
```

# PDE Example

# Discretize using Finite Difference Method

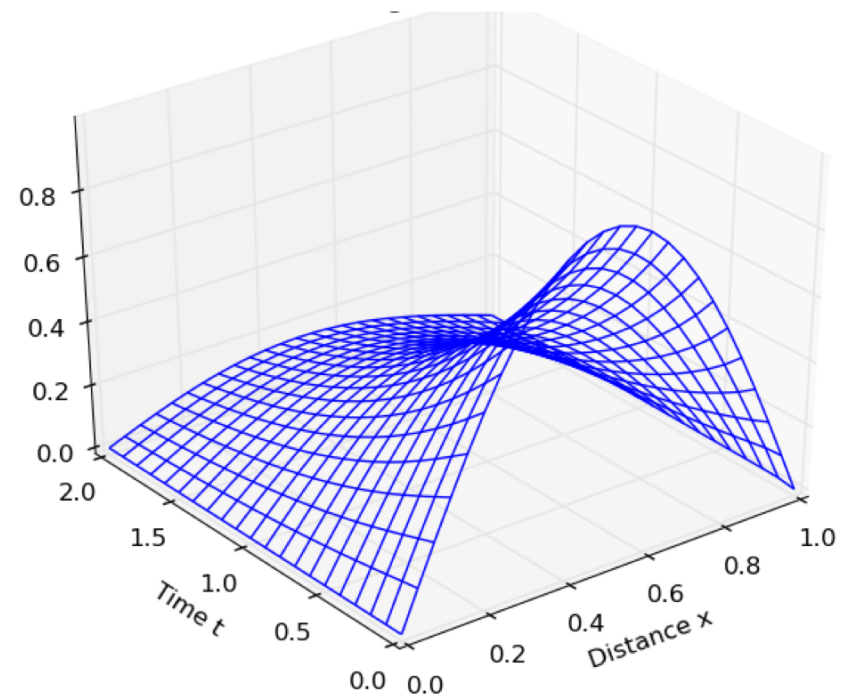
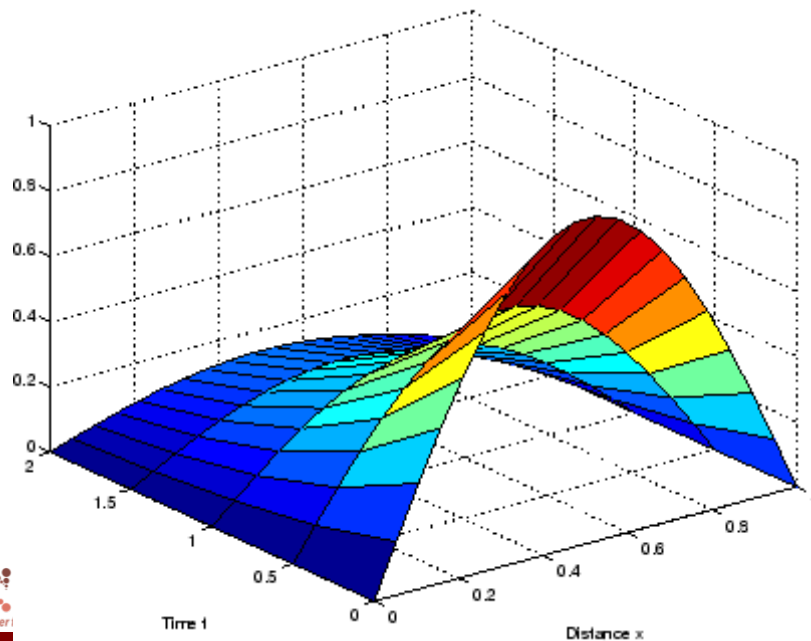
```
discretizer = TransformationFactory('dae.finite_difference')
```

```
discretizer.apply_to(m,nfe=25,wrt=m.x,scheme='BACKWARD')
```

```
discretizer.apply_to(m,nfe=20,wrt=m.t,scheme='BACKWARD')
```

```
solver = SolverFactory('ipopt')
```

```
results = solver.solve(m,tee=True)
```



# Upcoming Features

---

- Interface to integrator/DAE solver for:
  - Model simulation
  - Model initialization
  - Implementation of multiple shooting
- Interpolation tools for:
  - Model initialization



# Building Expert Frameworks with pyomo.dae Sandia National Laboratories

---

- Nonlinear model predictive control
- Reaction kinetics

# Nonlinear Model Predictive Control

```
NMPC
```

```
from iNMPC import *
CONTROLLER = IdealNMPC(model=m, model_time=m.t, prediction_horizon=50,
 control_horizon=25, simulation_periods=50, default_discretization=True)
#CONTROLLER.discretize_model(discretization_type='OrthogonalCollocation',
ncp=5, wrt=m.t, scheme='LAGRANGE-LEGENDRE')
CONTROLLER.solve('ipopt', options=[('linear_solver', 'ma57')],
 file_name='test1', dynamic_plot=True)
```

```
asNMPC
```

```
from asNMPC import *
CONTROLLER = AdvancedStepNMPC(model=m, model_time=m.t, prediction_horizon=50,
 control_horizon=25, simulation_periods=50, default_discretization=True)
CONTROLLER.solve('ipopt_sens', options=[('linear_solver', 'ma57'),
 ('run_sens', 'yes')], file_name='test1', real_plant = plant,
 dynamic_plot=True)
```

[Control frameworks developed by Federico Lozano Santamaría Universidad de los Andes, Bogotá, Colombia]

# Reaction Kinetics

- Chemical reaction kinetics:  $2A \xrightarrow{k_1} B; \quad B \xrightarrow{k_2} C$
- Reaction rate:
 
$$r_1 = k_1 c_A^2 \quad k_1 = A_1 e^{\left(\frac{E_a}{R \cdot T}\right)}$$

$$r_2 = k_2 c_B \quad k_2 = A_2 e^{\left(\frac{E_a}{R \cdot T}\right)}$$
- Stoichiometry
 
$$\frac{dc_A}{dx} = -2r_1 k_1$$

$$\frac{dc_B}{dx} = r_1 k_1 - r_2 k_2$$

$$\frac{dc_C}{dx} = r_2 k_2$$

# General purpose kinetic model

```
def create_kinetic_model(rxnNet, time):
 model = ConcreteModel()
 model.SPECIES = Set(initialize=rxnNet.species())
 model.REACTIONS = Set(initialize=rxnNet.reactions.keys())
 model.TIME = ContinuousSet(bounds=(0,max(time)), initialize=time)
 model.c = Var(model.TIME, model.SPECIES, bounds=(0,None))
 model.dcdt = DerivativeVar(model.c, wrt=model.TIME)
 model.k = Var(model.REACTIONS, bounds=(0,None))
 model.rate = Var(model.TIME, model.REACTIONS)
 def reaction_rate(m, t, r):
 rhs = m.k[r]
 for s, coef in iteritems(m.rxnNetwork.reactions[r].reactants):
 rhs *= m.c[t,s]**coef
 return m.rate[t,r] == rhs
 model.reaction_rate = Constraint(model.TIME, model.REACTIONS, rule=reaction_rate)
 def stoichiometry(m, t, s):
 rhs = 0
 for r in m.REACTIONS:
 if s in m.rxnNetwork.reactions[r].reactants:
 rhs -= m.rate[t,r] * m.rxnNetwork.reactions[r].reactants[s]
 if s in m.rxnNetwork.reactions[r].products:
 rhs += m.rate[t,r] * m.rxnNetwork.reactions[r].products[s]
 return m.dcdt[t,s] == rhs
 model.stoichiometry = Constraint(model.TIME, model.SPECIES, rule=stoichiometry)
 return model
```

# Step 1: simulation

```
fdiff = TransformationFactory("dae.finite_difference")
rxns = ReactionNetwork()
rxns.add(Reaction("AtoB", reactants=['2*A'], products=['B']))
rxns.add(Reaction("BtoC", reactants=['B'], products=['C']))
```

```
model = create_kinetic_model(rxns, 60*60)
```

```
A1 = 1.32e19 # L / mol*s
```

```
A2 = 1.09e13 # 1/s
```

```
Ea1 = 140000 # J/mol
```

```
Ea2 = 100000 # J/mol
```

```
R = 8.314 # J / K*mol
```

```
T = 330 # K
```

```
model.k['AtoB'].fix(A1 * exp(-Ea1 / (R*T)))
```

```
model.k['BtoC'].fix(A2 * exp(-Ea2 / (R*T)))
```

```
model.c[0, 'A'].fix(1)
```

```
model.c[0, 'B'].fix(0)
```

```
model.c[0, 'C'].fix(0)
```

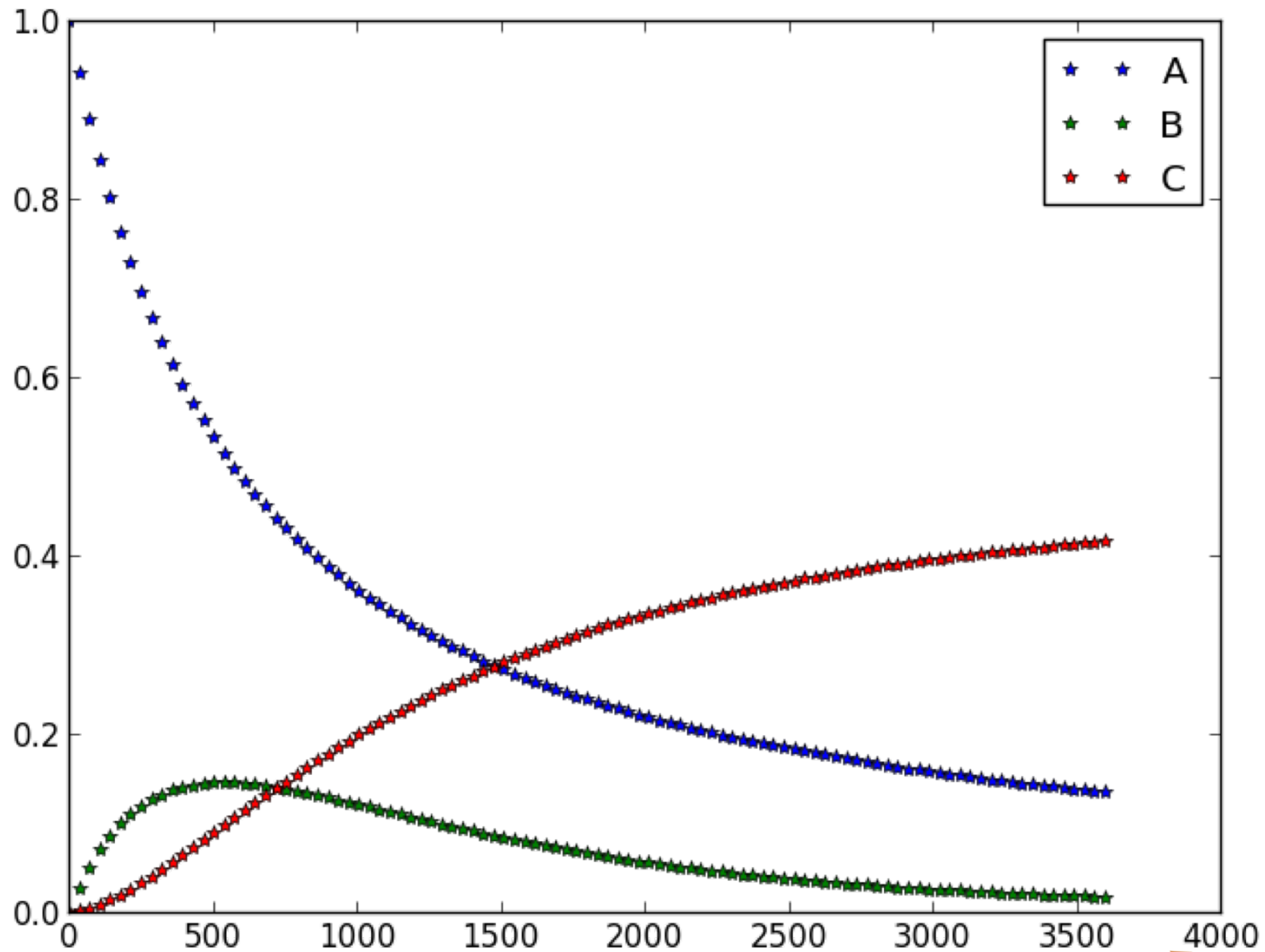
```
fdiff.apply_to(model, nfe=100)
```

```
solver.solve(model)
```

```
plot_results(model)
```

```
def plot_results(model):
 if plt is not None:
 _tmp = sorted(iteritems(model.c))
 for _i, _x in enumerate('ABC'):
 plt.plot([x[0][0] for x in _tmp if x[0][1] == _x],
 [value(x[1]) for x in _tmp if x[0][1] == _x],
 'bgr'[_i]+'*', label=_x)
 plt.legend()
 plt.show()
```

# Step 1: results



# Step 2: maximize the production of “B”

```
fdiff = TransformationFactory("dae.finite_difference")
rxns = ReactionNetwork()
rxns.add(Reaction("AtoB", reactants=['2*A'], products=['B']))
rxns.add(Reaction("BtoC", reactants=['B'], products=['C']))
model = create_kinetic_model(rxns, 60*60)

A1 = 1.32e19 # L / mol*s
A2 = 1.09e13 # 1/s
Ea1 = 140000 # J/mol
Ea2 = 100000 # J/mol
R = 8.314 # J / K*mol
```

```
model.T = Var(bounds=(0,None), initialize=330) # K
def compute_k(m):
 yield m.k['AtoB'] == A1 * exp(-Ea1 / (R*m.T))
 yield m.k['BtoC'] == A2 * exp(-Ea2 / (R*m.T))
model.compute_k = ConstraintList(rule=compute_k)
```

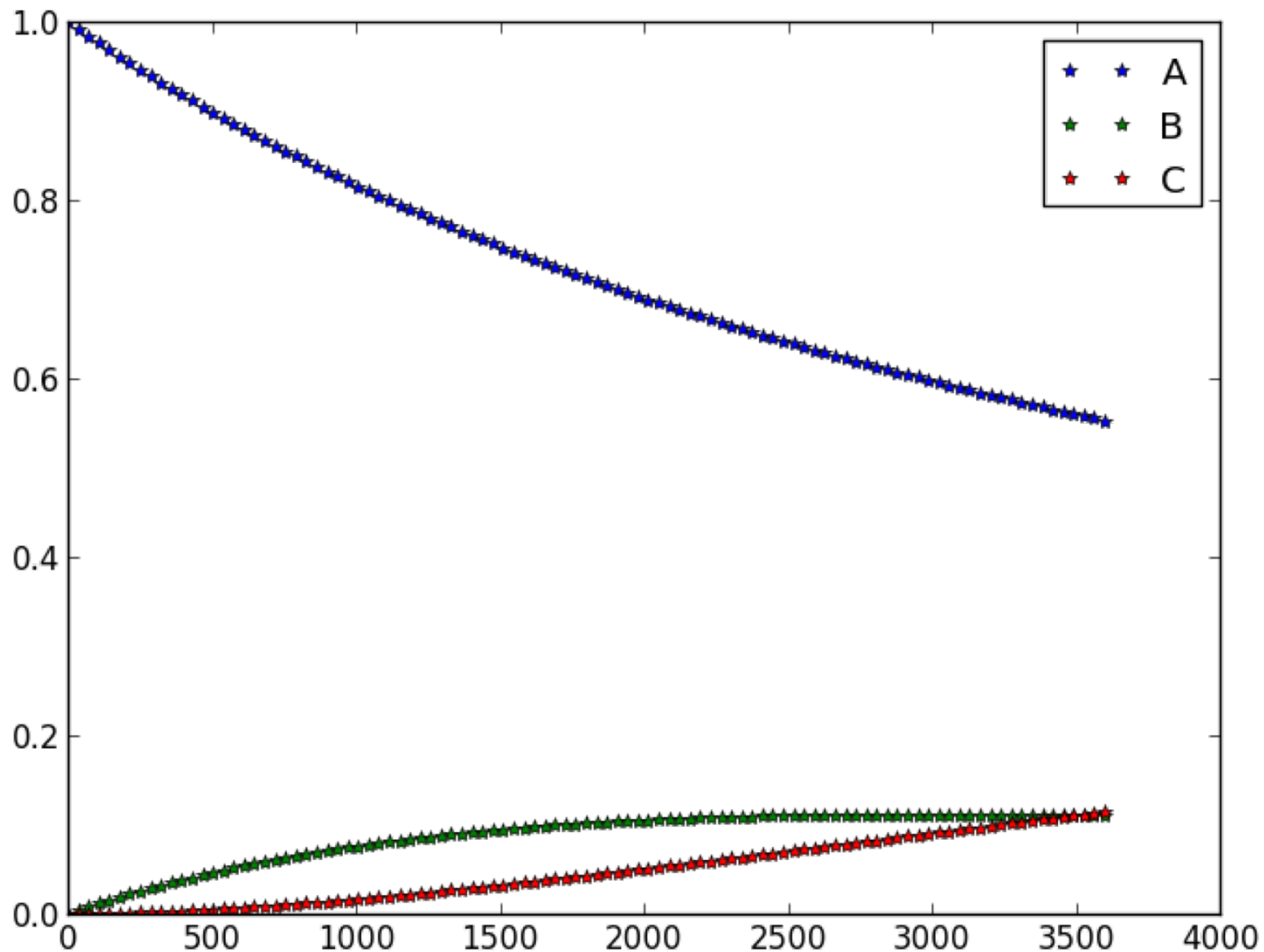
```
model.c[0, 'A'].fix(1)
model.c[0, 'B'].fix(0)
model.c[0, 'C'].fix(0)

fdiff.apply_to(model, nfe=100)
```

```
model.obj = Objective(sense=maximize, expr=model.c[max(model.TIME), 'B'])
```

```
solver.solve(model)
plot_results(model)
```

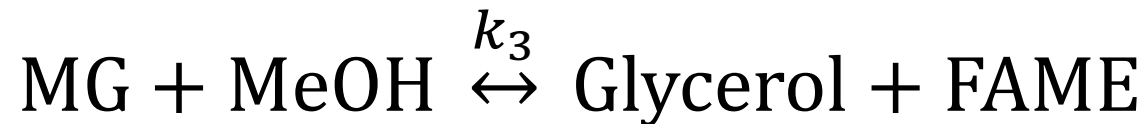
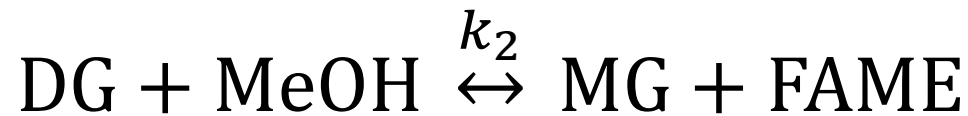
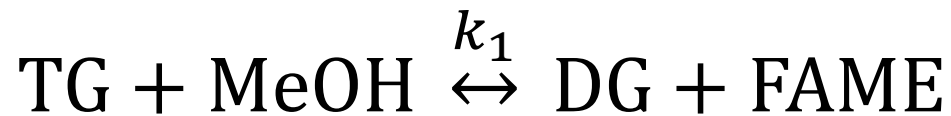
## Step 2: maximize the production of “B”





# Kinetic parameter regression

- (Assumed) Chemical reaction kinetics:



- Given experimental data, estimate the reaction rate constants

# Generate (sub)model for each experiment...

```
def create_regression_model(b, t):
 rxns = ReactionNetwork()
 rxns.add_reversible(
 Reaction("k_1", reactants=['TG','MeOH'], products=['DG','FAME']))
 rxns.add_reversible(
 Reaction("k_2", reactants=['DG','MeOH'], products=['MG','FAME']))
 rxns.add_reversible(
 Reaction("k_3", reactants=['MG','MeOH'], products=['Glycerol','FAME']))

 data = b.model().data[t]
 key = b.model().key

 model = create_kinetic_model(rxns, data.keys())

 model.error = Var(bounds=(0,None))

 model.compute_error = Constraint(
 expr= model.error == sum(
 ((model.c[t,key[i]] - x) / max(data[_t][i] for _t in data))**2
 for t in data for i,x in enumerate(data[t])))

 return model
```

# Assemble the regression model and solve

```
model = ConcreteModel()
model.key = key = ('MeOH', 'TG', 'DG', 'MG', 'FAME', 'Glycerol')
model.data = data = { 150: { 0: (2.833,6.84E-02,0.00,0.00,0.00,0.00,),
 256: (2.807,4.75E-02,1.51E-02,3.71E-03,2.60E-02,8.18E-04,), # ...
 } 210: { # ...
 } }

model.experiment = Block(data.keys(), rule=create_regression_model)
model.obj = Objective(sense=minimize,
 expr=sum(b.error for b in model.experiment[:]))

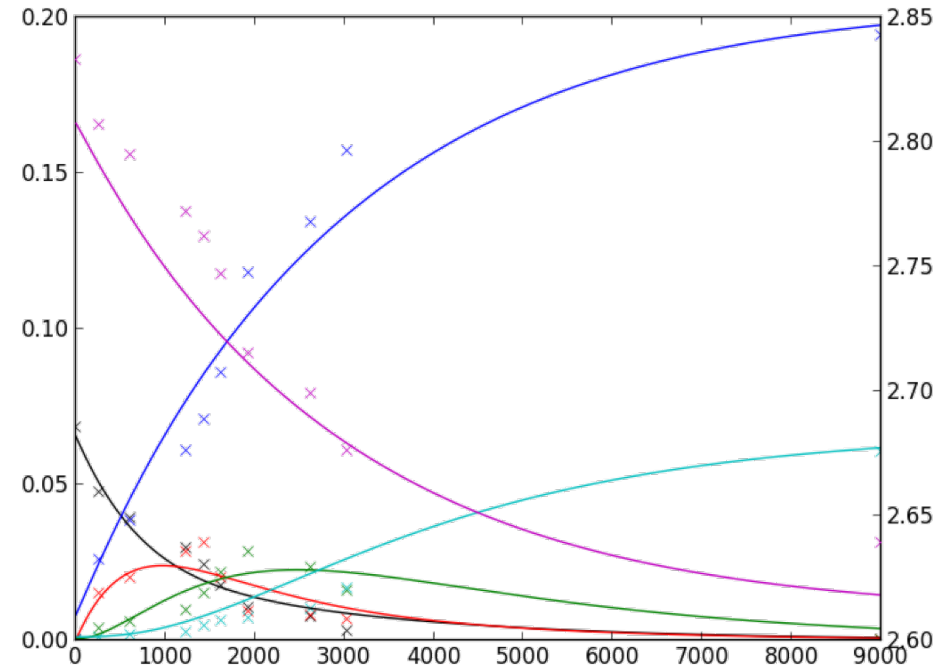
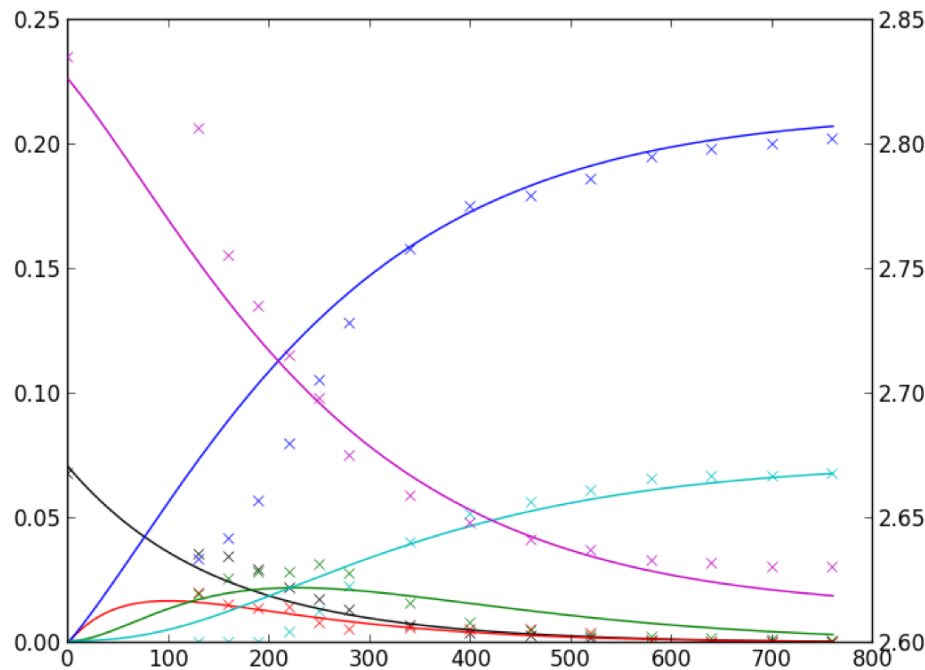
initializations from the paper
for exp in model.experiment[:]:
 exp.k['k_1'] = 7.58e-7
 exp.k['k_1_r'] = 0
 exp.k['k_2'] = 2.20e-7
 exp.k['k_2_r'] = 0
 exp.k['k_3'] = 2.15e-7
 exp.k['k_3_r'] = 0

colloc.apply_to(model, nfe=100, ncp=3)
solver.solve(model, tee=True)
plot_regression_results(model)
```

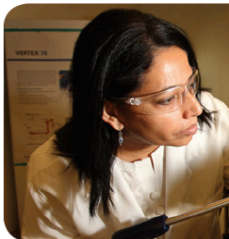
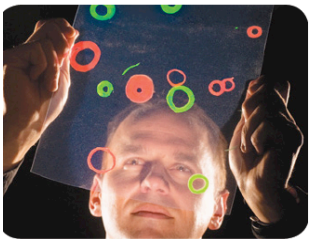
# Regression results

T=150

T=210



- 10850 variables, 10826 constraints
- Total time: 6.4 seconds
  - 2.8 seconds for model generation and processing
  - 3.6 seconds in solver (ipopt)



*Exceptional  
service  
in the  
national  
interest*

## 8. Stochastic Programming



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Competing on Analytics

|                       |                                |                                       |             |
|-----------------------|--------------------------------|---------------------------------------|-------------|
| Competitive Advantage | Optimization Under Uncertainty | How to handle incomplete information? | Decision    |
|                       | Optimization                   | How do we achieve the best outcome?   |             |
|                       | Predictive Modeling            | What will happen next if...?          | Prediction  |
|                       | Forecasting                    | What if these trends continue?        |             |
|                       | Simulation                     | What could happen...?                 |             |
|                       | Alerts                         | What actions are needed?              | Description |
|                       | Query/Drill Down               | What exactly is the problem?          |             |
|                       | Ad-hoc Reporting               | How many, how often, where?           |             |
|                       | Standard Reporting             | What happened?                        |             |

# Optimization Under Uncertainty

- At its core, Optimization Under Uncertainty (OUU) is decision-making in the face of incomplete information
- (Some) Sources of uncertainty:
  - **Models:** “early” system decisions, un-modeled inputs, missing phenomena, parameter values
  - **Product performance:** manufacturing variability
  - **Usage:** boundary conditions, demands, weather
- How do we deal with uncertainty:
  1. Ignore it (deterministic optimization)
  2. Safety margins (leaving money on the table)
  3. Careful case study selection (cannot remove inherent randomness)
  4. Sensitivity analysis (challenging for larger problems)
  5. Integrate it into our decision-making (e.g., OUU)

# Approaches to OUU

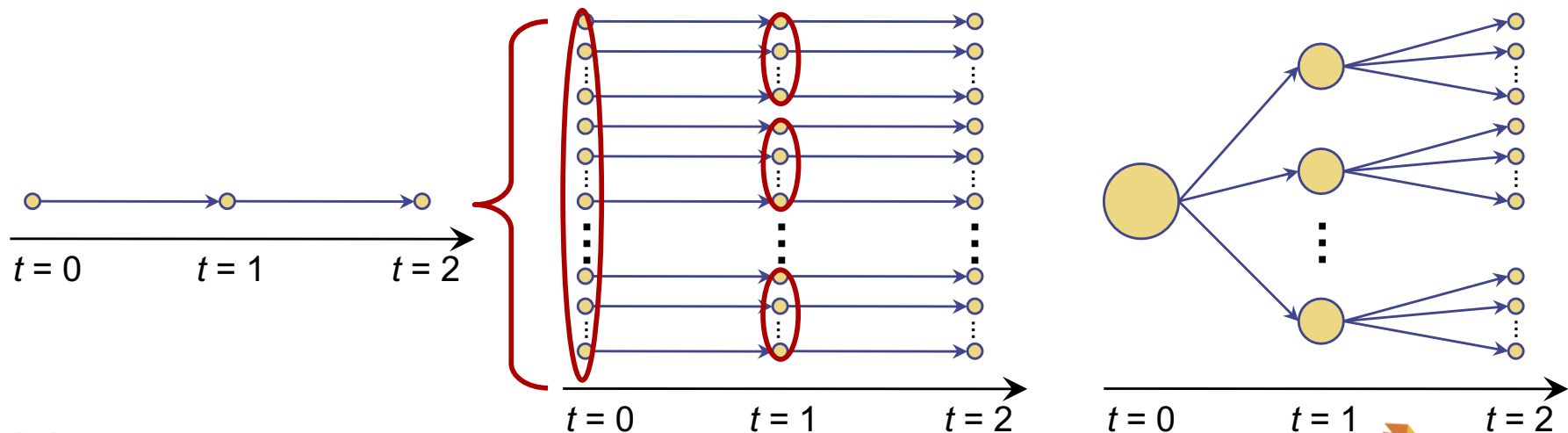
---

- Computational systems are essentially *deterministic*
  - OUU must be recast into a finite computable problem
- Common approaches:
  - Control the uncertainty
    - Feedback, feedforward, model predictive control (MPC), nonlinear MPC
    - Goal is to adapt the system (in real time) as the uncertainty is revealed
  - Bound the uncertainty
    - Robust optimization, Adjustable robust optimization
    - Bound the uncertainty space and then optimize for the worst case
  - Work backwards
    - Dynamic programming, Approximate dynamic programming
    - Compute a value for all possible states, working back from the final state
  - Sample the uncertainty
    - Sample average approximation, **Stochastic programming**
    - Sample the uncertain space and then optimize over the samples



# Stochastic Programming

- Given a potentially infinite uncertainty space
  - That may not be characterized by distributions
  - That may be heavily correlated (in space, time, or function)
- *Approximate* the uncertainty by selecting a finite set of *scenarios*
  - Scenarios may not all be equally likely
  - We could optimize each scenario separately (compute the *value of perfect information*)
    - But I must implement something now... what should I do?
  - Solve a single large problem with scenario decisions constrained to be *implementable*;
    - For a given decision, scenarios that are indistinguishable must all choose the same decision.
    - *Nonanticipativity constraints*



# Using PySP: Overview

---

- PySP simplifies the process of formulating Stochastic Programs by automating the bookkeeping and model construction process.
- This generally reduces to the following four steps:
  1. Formulate the deterministic model
  2. Specify the deterministic model data
    - (necessary to *debug* the deterministic model!)
  3. Specify the scenario tree
  4. Specify the scenario instance data
- Important:
  - We are only talking about instance specification – not solvers

# Example: Production Planning

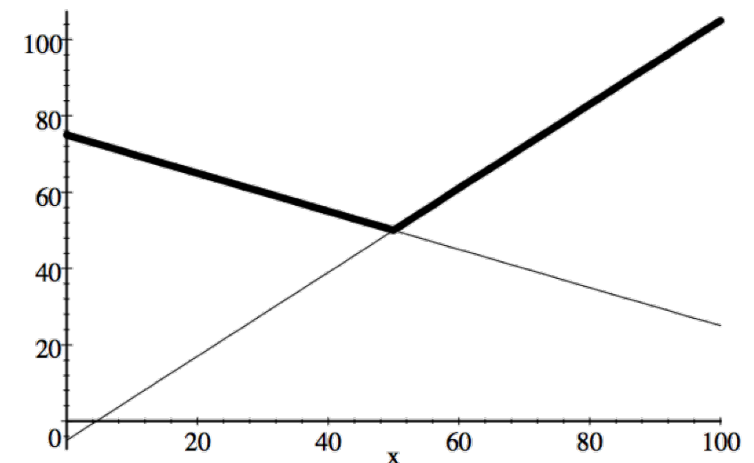
Example from Alexander Shapiro and Andy Philpott (2007)

A company has decided to order a quantity  $x$  of a product to satisfy demand  $d$ . The per-unit cost of ordering is  $c$ . If demand  $d$  is greater than  $x$ , then the back-order penalty is  $b$  per unit. If demand is less than the order quantity, i.e.,  $d < x$ , 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:  $c=1$ ,  $b=1.5$ ,  $h=0.1$ ,  $d=50$



# Deterministic Formulation

---

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.

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

We can formulate a two-stage stochastic program where the first stage has zero cost and the second stage has cost  $t$ .

# Deterministic Formulation in Pyomo

```
from pyomo.environ import *

c, b, h = 1.0, 1.5, 0.1

model = AbstractModel()
model.d = Param(within=NonNegativeReals)
model.t = Var()
model.x = Var(within=NonNegativeReals, bounds=(0,100))

model.c1 = Constraint(expr= model.t >= (c-b)*model.x + b*model.d)
model.c2 = Constraint(expr= model.t >= (c+h)*model.x - h*model.d)

model.FirstStageCost = Expression(expr=0)
model.SecondStageCost = Expression(expr=model.t)
model.obj = Objective(expr=model.FirstStageCost + model.SecondStageCost)
```

# Deterministic Model Data

---

- We used native Python data for the values of  $c$ ,  $b$ , and  $h$ 
  - These parameters don't vary across scenarios
- We can put the scenario-dependent parameters into a distinct .dat file (for example) that contains the single line:
  - `param d := 50 ;`
- For pedagogical purposes, we won't use a concrete model quite yet...
  - Avoids the need for scenario-specific data files

# Specifying the Scenario Tree: *names*

- Under the hood, a *Pyomo Model* is used to specify a Scenario tree
  - The scenario tree can be specified through a standard DAT file

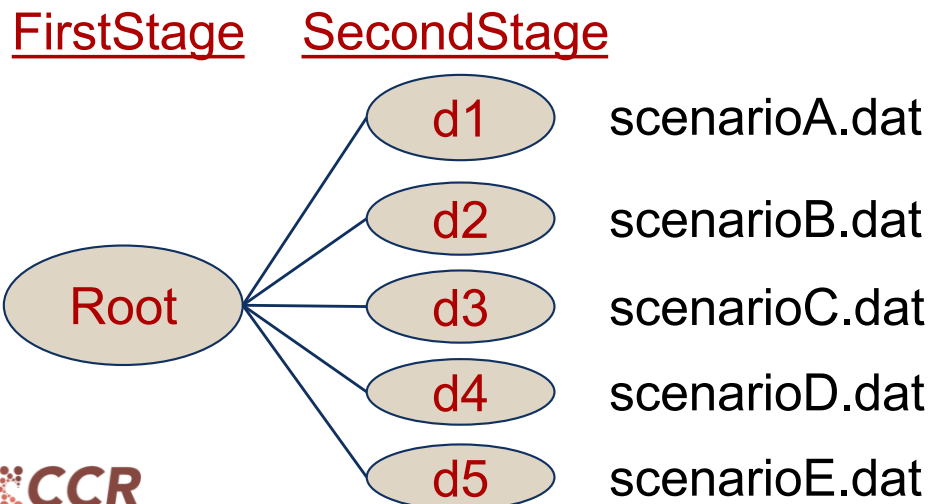
```
model = AbstractModel()

model.Stages = Set(ordered=True)

model.Nodes = Set(ordered=True)
```

```
set Stages := FirstStage
 SecondStage;

set Nodes := Root
 d1 d2 d3 d4 d5 ;
```



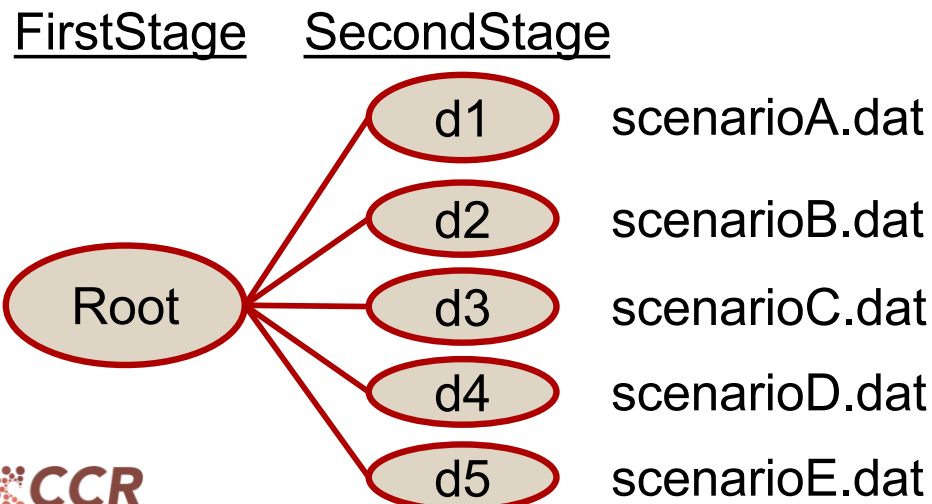
# Specifying the Scenario Tree: *structure*

```
model.NodeStage = Param(
 model.Nodes,
 within=model.Stages)
```

```
model.Children = Set(
 model.Nodes,
 within=model.Nodes,
 ordered=True)
```

```
param NodeStage :=
 Root FirstStage
 d1 SecondStage
 d2 SecondStage
 d3 SecondStage
 d4 SecondStage
 d5 SecondStage ;
```

```
set Children[Root] :=
 d1 d2 d3 d4 d5 ;
```

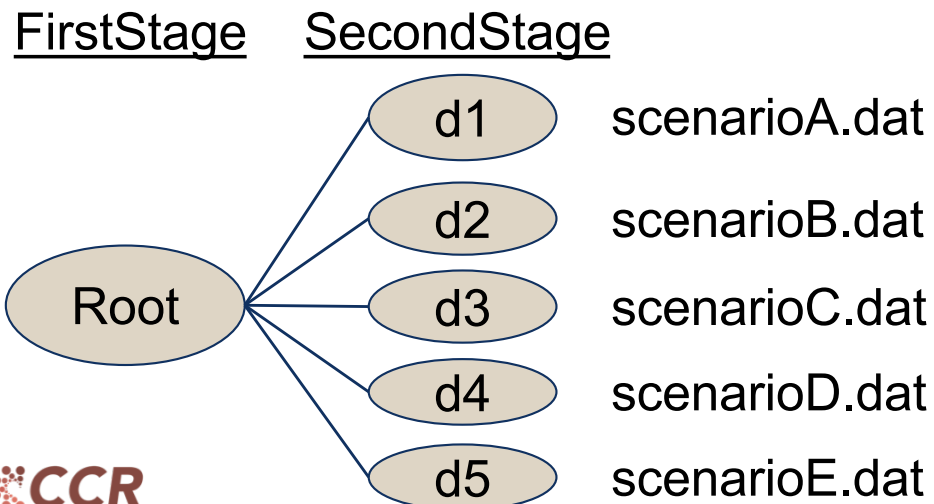




# Specifying the Scenario Tree: *probabilities*

```
model.ConditionalProbability =
 Param(model.Nodes)
```

```
param
ConditionalProbability :=
 Root 1.0
 d1 0.2
 d2 0.2
 d3 0.2
 d4 0.2
 d5 0.2 ;
```



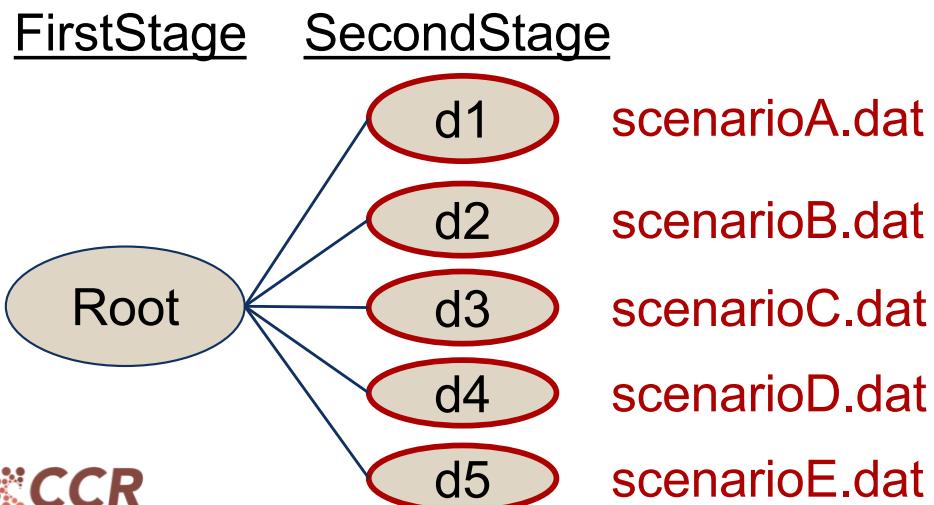
# Specifying the Scenario Tree: *scenarios*

```
model.Scenarios = Set(
 ordered=True)
```

```
model.ScenarioLeafNode = Param(
 model.Scenarios,
 within=model.Nodes)
```

```
set Scenarios :=
 scenarioA
 scenarioB
 scenarioC
 scenarioD
 scenarioE ;
```

```
set ScenarioLeafNode :=
 scenarioA d1
 scenarioB d2
 scenarioC d3
 scenarioD d4
 scenarioE d5 ;
```



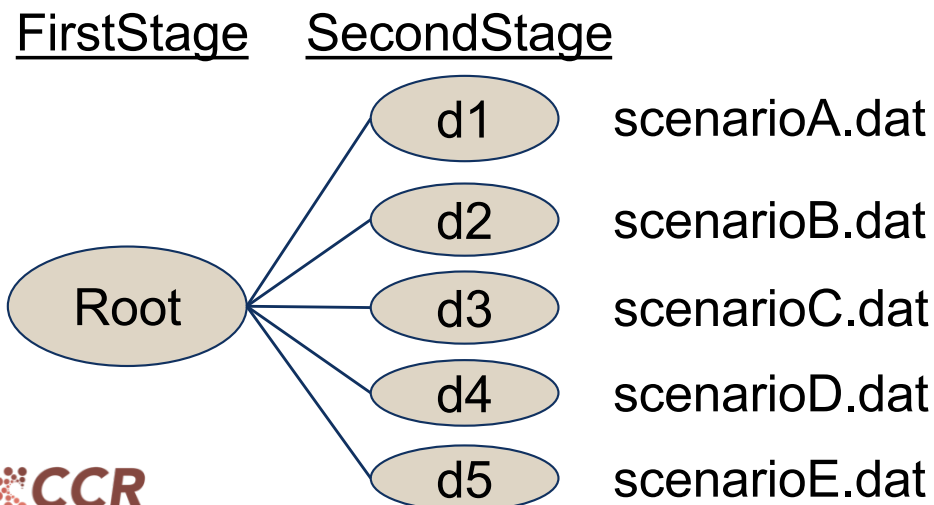
# Specifying the Scenario Tree: *variables*

- PySP needs to know which variables belong to which stages
  - For forming the nonanticipativity constraints
  - Can use slicing and wildcards (e.g., “x[\*]” to indicate all indices of “x”)

```
model.StageVariables = Set(
 model.Stages,
 ordered=True)
```

```
set
StageVariables[FirstStage]
 := x ;

set
StageVariables[SecondStage]
 := t ;
```

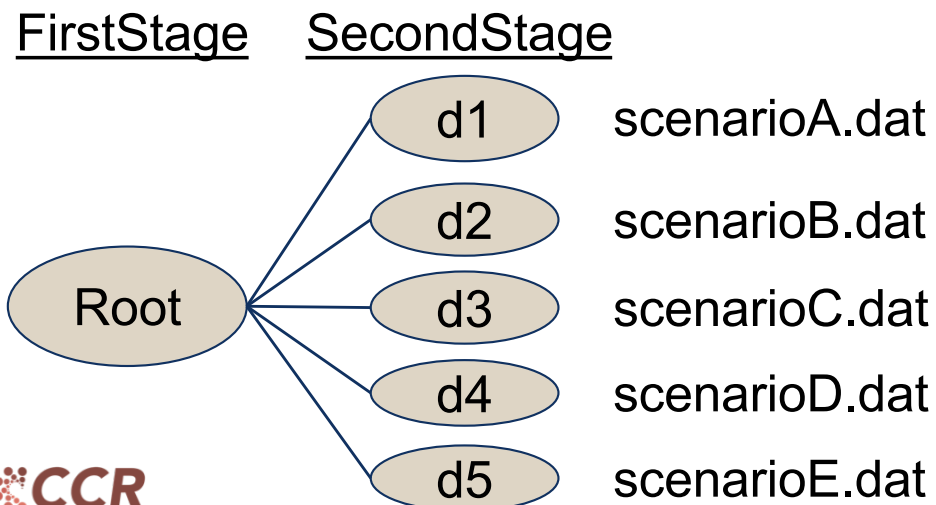


# Specifying the Scenario Tree: *objective*

- PySP needs to know the cost of each stage
  - Frequently a stage cost may be “0” (as it is in the newsvendor)
  - The stage costs are frequently **Expressions**

```
model.StageCost = Param(
 model.Stages)
```

```
param StageCost :=
 FirstStage FirstStageCost
 SecondStage SecondStageCost
```



# Specifying the Scenario Tree: *advanced topics*

```
model.NodeVariables = Set(
 model.Nodes, ordered=True)

model.StageDerivedVariables = Set(
 model.Stages, ordered=True)

model.NodeDerivedVariables = Set(
 model.Nodes, ordered=True)

model.ScenarioBasedData = Param(
 within=Boolean, default=True)

model.Bundling = Param(
 within=Boolean, default=False)

model.Bundles = Set(ordered=True)

model.BundleScenarios = Set(
 model.Bundles, ordered=True)
```

- You can specify variable locations by node as well as stage
- Frequently a subset of stage variables are “derived” – that is, completely calculable from other stage variables. This avoids unnecessary NACs.
- Data can be provided “per node” instead of “per scenario”
- Scenarios may be bundled together by
  - Specifying the bundle names
  - Assigning scenarios to bundles
- Bundles can also be created by randomly assigning scenarios

# Scenario Data in PySP

---

- Two methods are available to specify scenario-specific data
  - Scenario-based
  - Node-based
- In the scenario-based approach, a single and complete .dat file is specified for each individual scenario
  - Redundant, but straightforward if computer-generated
- In the node-based approach, a single .dat file is specified for each node in the scenario tree
  - Maximally compact, but requires some book-keeping
- This example uses scenario-based data, with data files that simply define  $d$ .

# Solving the Extensive Form (1)

---

- In PySP, the `runef` script is provided to create, write, and solve the extensive form of a stochastic programming model
- The basic command-line:
  - `runef -m models -i scenarios --solve \`  
`--solver=glpk`
  - Assumes that our deterministic model is in a file called `ReferenceModel.py` in the `models` sub-directory
  - Assumes scenario-specific `.dat` files are in the `scenarios` sub-directory
- NOTE: Even commercial solvers often have difficulty solving EFs
- NOTE: The `runef` script is agnostic to the details of the EF
  - Integers, Binaries, DAEs, disjunctions, bi-level, ...

# Solving the Extensive Form (2)

- After the solution, runef reports information about scenario tree solution
  - Variable values
  - Costs
- Also accessible via an API
  - That is not presently documented...

## Tree Nodes:

```

Name=RootNode
Stage=FirstStage
Parent=None
Conditional probability=1.0000
Children:
 d1
 d2
 d3
 d4
 d5
Scenarios:
 s1
 s2
 s3
 s4
 s5
Expected cost of (sub)tree rooted at node= 76.5000

Name=d1
Stage=SecondStage
Parent=RootNode
Conditional probability=0.2000
Children:
 None
Scenarios:
 s1
Expected cost of (sub)tree rooted at node= 64.5000

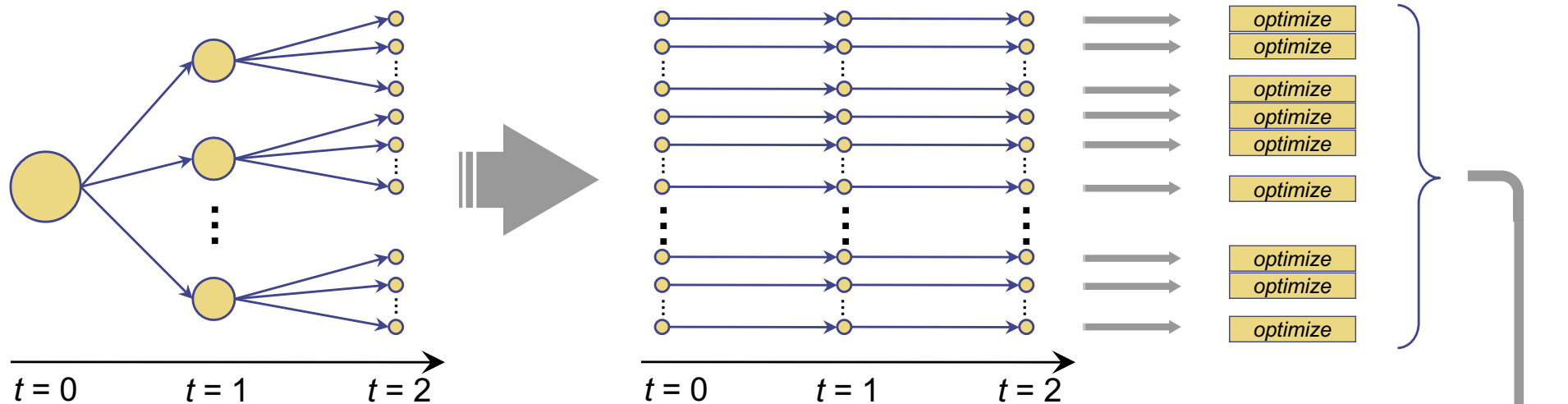
Name=d2
Stage=SecondStage
Parent=RootNode
Conditional probability=0.2000
Children:
 None

```



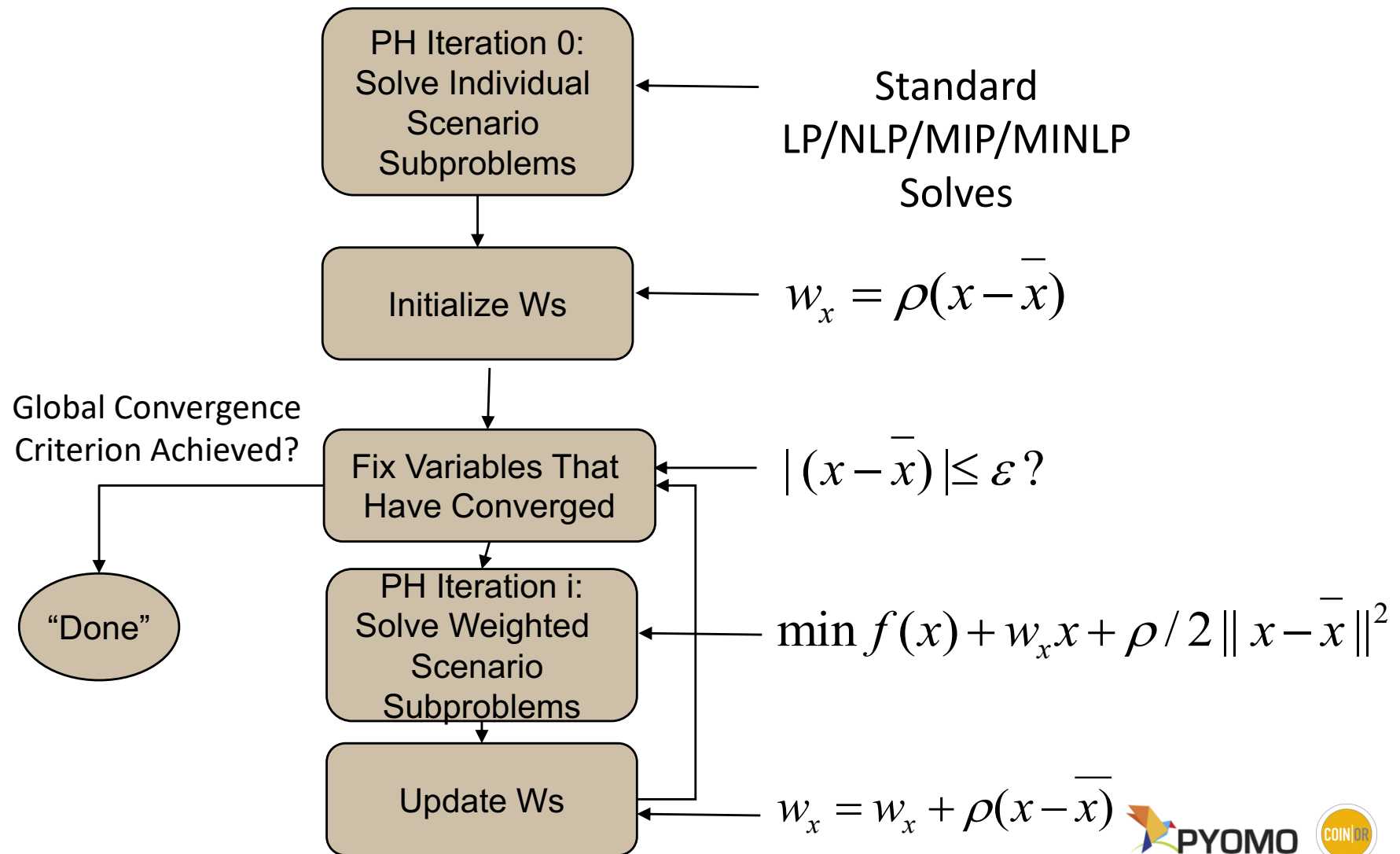
# What if the Problem is Too Difficult?

- Use problem decomposition!
  - Stage-wise (e.g., Benders decomposition [Benders, 1962])
    - Master problem (1<sup>st</sup> stage), independent (2<sup>nd</sup> stage) subproblems
    - Master problem grows with cuts from subproblems
  - Scenario-based (e.g., Progressive Hedging [Rockafellar & Wets, 1991])
    - No master problem
    - Iteratively converge NACs by penalizing deviation from consensus



# Progressive Hedging Algorithm

**Idea:** Use scenario-based decomposition



# Progressive Hedging (1)

---

- The execution of PH requires the specification of the penalty parameter ( $\rho$ )
- A global  $\rho$  value can be easily specified:
  - `runph -m models -i scenarios --default-rho=0.05`
- The quadratic penalty term in PH can be computationally problematic
  - E.g., quadratic MIP solvers can be 10x or slower than MIP solvers
  - Open-source quadratic solvers are (almost) non-existent
- PySP provides automatic, generic linearization mechanisms
  - Requires specification of variable lower and upper bounds
  - Specify number of breakpoints, distribution strategy
  - `runph -m models -i scenarios \`  
`--solver=glpk --default-rho=0.05 \`  
`--linearize-nonbinary-penalty-terms=100`

# Progressive Hedging (2)

---

- In the presence of integers, PH is no longer guaranteed to converge
  - Cycling behavior
  - Stagnation behavior
- To facilitate PH convergence for mixed-integer stochastic programs, PySP provides various configurable mechanisms
  - “Watson-Woodruff” Extensions
    - Computational Management Science (2009)
  - Implemented via a generic plug-in callback framework
- Capabilities include:
  - Variable fixing
  - Cycle detection
  - Cycle breaking
  - Slamming

# Parallelism for Free in PySP / PH

- PH is widely executed in parallel, on medium-scale clusters and many-core SMP workstations
- Parallelism in Pyomo (and therefore PySP) is based on Pyro
  - Straightforward master-slave or RPC-style communication
- Requires the following “helper” processes
  - `pyomo_ns` – the name server
  - `dispatch_srvr` – routes requests to perform work to workers
  - `phsolverserver` – responsible for manipulation and solution of specific scenario sub-problems
- Command-line is then as follows (in the SMP case):
  - `mpiexec -np 1 pyomo_ns : -np 1 dispatch_srvr :  
-np X phsolverserver : -np 1 runph --solver-  
manager=phpyro ...`

# Model Construction via Callbacks in PySP

- Scenario models can also be specified via callbacks in the model file supplied to PySP scripts (e.g., `runef` and `runph`)
  - Allows for the use of `ConcreteModel` alternatives
  - Avoids the need for scenario-specific `.dat` files
- Callback signature is as follows:

```
def pysp_instance_creation_callback(scenario_name, node_names):
 # load the data and build some_model...
 return some_model
```
- From the scenario and node names, the end-user can construct and return an arbitrary scenario model

# Scenario Tree Definition via Callbacks (1)

- Scenario trees can also be specified via callbacks in the model file supplied to PySP scripts (e.g., `runef` and `runph`)

- Signature:

```
def pysp_scenario_tree_model_callback():
 # load data and build scenario_tree_model...
 return scenario_tree_model
```

- The simplest approach is to import the Abstract model and create an instance:

```
from pyomo.pysp.scenariotree.tree_structure_model \
 import CreateAbstractScenarioTreeModel
def pysp_scenario_tree_model_callback():
 st = CreateAbstractScenarioTreeModel().create_instance()
 # ...
 return st
```

# Scenario Tree Definition via Callbacks (2)

- Constructing the scenario tree for the News Vendor example:

```
from pyomo.pysp.scenariotree.tree_structure_model import CreateAbstractScenarioTreeModel
def pysp_scenario_tree_model_callback():
 m = CreateAbstractScenarioTreeModel().create_instance()
 m.Stages.add('FirstStage')
 m.Stages.add('SecondStage')
 m.Nodes.add('Root')
 m.NodeStage['Root'] = 'FirstStage'
 m.ConditionalProbability['Root'] = 1
 for i in range(5):
 node = 'd%s' % (i+1,)
 m.Nodes.add(node)
 m.NodeStage[node] = 'SecondStage'
 m.Children['Root'].add(node)
 m.ConditionalProbability[node] = 1./N
 scenario = 'scenario%s' % (chr(ord('A')+i),)
 m.Scenarios.add(scenario)
 m.ScenarioLeafNode[scenario] = node
 m.StageVariables['FirstStage'].add('x')
 m.StageVariables['SecondStage'].add('t')
 m.StageCost['FirstStage'] = 'FirstStageCost'
 m.StageCost['SecondStage'] = 'SecondStageCost',
 return m
```



# Scenario Tree definition via callbacks (3)

- Two-stage models are so common that we provide a helper function:

```
from pyomo.pysp.scenariotree.tree_structure_model import \
 CreateConcreteTwoStageScenarioTreeModel
def pysp_scenario_tree_model_callback():
 m = CreateConcreteTwoStageScenarioTreeModel(5)
 m.StageVariables[m.Stages.first()].add('x')
 m.StageVariables[m.Stages.last()].add('t')
 m.StageCost[m.Stages.first()] = 'FirstStageCost'
 m.StageCost[m.Stages.last()] = 'SecondStageCost',
 return m
```

- Note that the Scenarios and Nodes are named differently relative to our initial example
  - E.g., scenarios are now named “Scenario1”, “Scenario2”, ...

# The PySP Solver Zoo

- We only regularly advertise two solvers in PySP
  - `runef` and `runph`
- But, we also have implementations and/or interfaces to a wide variety of alternatives
  - Benders / L-shaped
    - `$ python -m pyomo.pysp.solvers.benders`
  - BBPH (Branch-and-Bound / PH hybrid)
    - `$ python -m pyomo.pysp.solvers.bbph`
  - DDSIP (Dual Decomposition)
    - `$ python -m pyomo.pysp.solvers.ddsip`
  - Stochastic Decomposition (Sen et al.)
    - `$ python -m pyomo.pysp.solvers.sd`
  - SchurIpopt (Laird et al.)
    - `$ python -m pyomo.pysp.solvers.schuripopt`

**NOTE:** Many of these solvers are on the `pysp_solver_interface` branch, and are due to be merged onto master Q3CY17.

# The End

---

Questions?

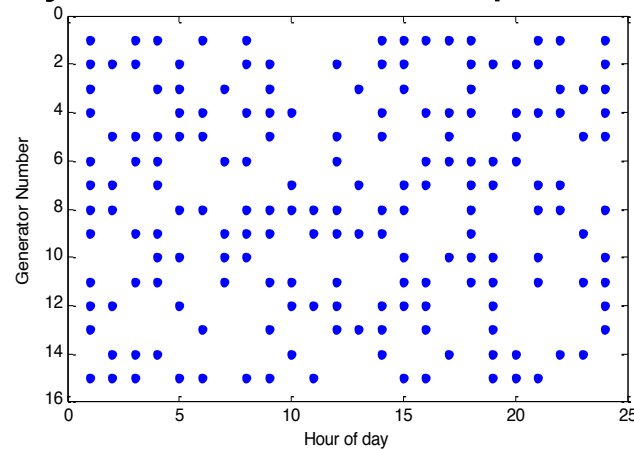
# Power Grid: A Rich Application Space

---

- Power grid operations with high-penetration renewables
  - Stochastic economic dispatch
  - Stochastic unit commitment
  - Contingency analysis
- Transmission / generation expansion planning
- Common theme: operate a large scale system with discrete decisions with high reliability in the face of uncertain {demands, generation, configurations} and at low cost.

# Stochastic Unit Commitment (At Scale)

Objective: Minimize expected cost



First stage variables:

- Unit On / Off



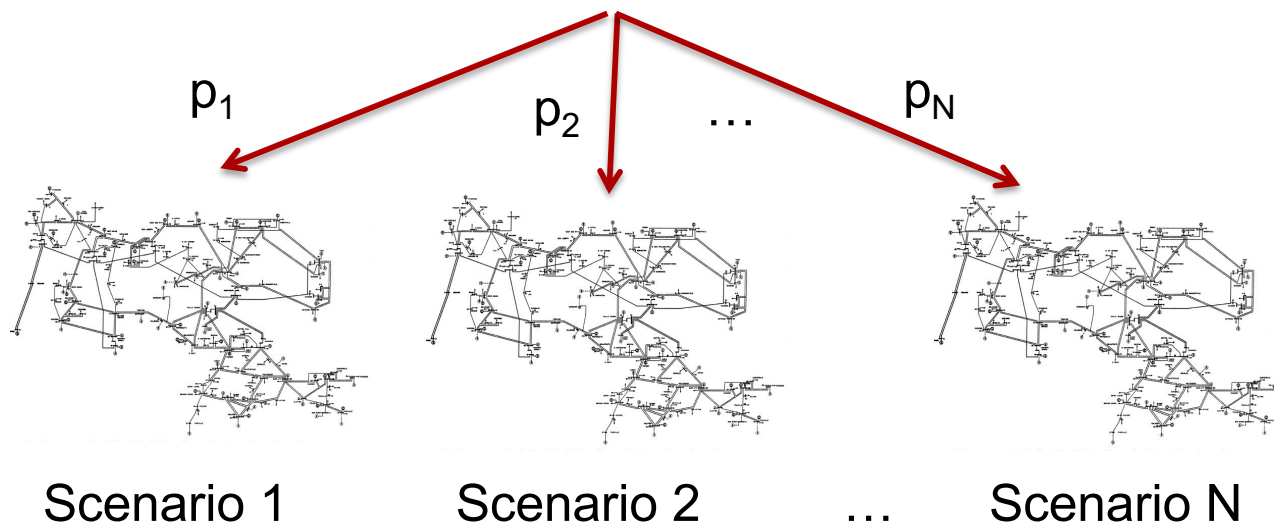
Nature resolves uncertainty

- Load
- Renewables output
- Forced outages



Second stage variables  
(*per time period*):

- Generation levels
- Power flows
- Voltage angles
- ...



# Solving the Stochastic Extensive Form

- Reliability Unit Commitment (RUC) Test Instance: WECC-240++
- J.E. Price, Reduced Network Modeling of WECC as a Market Design Prototype, 2011 IEEE PES General Meeting
- Changes necessary to create viable RUC test case
  - Addition of realistic ramping rates and min up/down time constraints
- Results

**Table 3** Solution quality statistics for the extensive form of the *WECC-240-r1* instance, given 4 hours of run time.

| # Scenarios | Objective Value | MIP Lower Bound | Gap % | Run Time (s) |
|-------------|-----------------|-----------------|-------|--------------|
| 3           | 64278.20        | 63797.72        | 0.75  | 14491        |
| 5           | 62740.67        | 62180.86        | 0.89  | 14723        |
| 10          | 61563.10        | 60835.45        | 1.18  | 14630        |
| 25          | 61455.55        | 59963.78        | 2.36  | 14960        |
| 50          | 61911.74        | 59540.87        | 3.83  | 15480        |
| 100         | 62388.85        | 59548.23        | 4.51  | 16562        |

# Progressive Hedging Results: WECC-240++

**Table 7** Solve time (in seconds) and solution quality statistics for PH executing on the *WECC-240-r1* instance, with  $\alpha = 0.5$ ,  $\mu = 6$ , and  $\gamma = 0.025$

| # Scenarios                 | Convergence Metric | Obj. Value | PH L.B.   | # Vars Fx. | Time      |
|-----------------------------|--------------------|------------|-----------|------------|-----------|
| 64-Core Workstation Results |                    |            |           |            | Latest... |
| 3                           | 0.0 (20 iters)     | 64213.397  | 63235.381 | 4080       | 508 166   |
| 5                           | 0.0 (in 18 iters)  | 62642.531  | 61767.253 | 4079       | 674 119   |
| 10                          | 0.0 (in 35 iters)  | 61396.553  | 60476.604 | 4066       | 648 167   |
| 25                          | 0.0 (in 22 iters)  | 60935.040  | 59992.622 | 4066       | 761 212   |
| 50                          | 0.0 (in 15 iters)  | 60625.149  | 59631.839 | 4034       | 1076 280  |
| 100                         | 0.0 (in 25 iters)  | 61155.387  | 60014.571 | 4080       | 1735 315  |
| Red Sky Results             |                    |            |           |            |           |
| 50                          | 0.0 (in 16 iters)  | 60623.343  | 59779.813 | 4007       | 404       |
| 100                         | 0.0 (in 25 iters)  | 61120.943  | 60275.744 | 4080       | 549       |

ISO-NE results are obtained on Red Sky on average in 10 minutes,  
20 minutes in the worst case (with 100 scenarios)

# Improved UC Formulations?

- Morales-Espana et al. (2013)
  - Extends prior tight formulation by Ostrowski et al.
- Shows off advantage of PH, in that improved deterministic models immediately impact stochastic solve times
- Results

**Table 10** Solve time (in seconds) and solution quality statistics for PH executing on the *WECC-240-r1* instance, with  $\alpha = 0.5$ ,  $\mu = 3$ , and the MTR deterministic UC model.

| # Scenarios                 | Convergence Metric | Obj. Value | PH L.B.   | # Vars Fx. | Time |
|-----------------------------|--------------------|------------|-----------|------------|------|
| 64-Core Workstation Results |                    |            |           |            |      |
| 3                           | 0.0 (in 36 iters)  | 64141.771  | 64109.021 | 4080       | 237  |
| 5                           | 0.0 (in 23 iters)  | 62628.532  | 62499.212 | 4080       | 161  |
| 10                          | 0.0 (in 26 iters)  | 61384.016  | 61327.734 | 4080       | 215  |
| 25                          | 0.0 (in 41 iters)  | 60927.903  | 60850.717 | 4080       | 366  |
| 50                          | 0.0 (in 11 iters)  | 60617.311  | 60470.956 | 4044       | 318  |

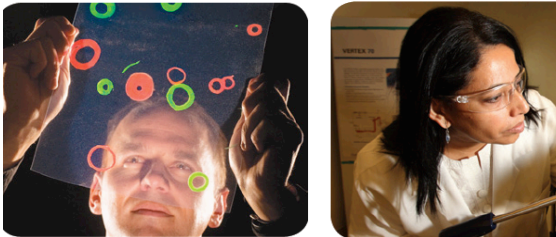
- ISO-NE results drop to 15 minutes maximum (10 average)



# Parallelization and Bundling

---

- Progressive Hedging is, at least conceptually, easily parallelized
  - Scenario sub-problem solves are clearly independent
  - Advantage over Benders, in that “bloat” is distributed
    - Critical in low-memory-per-node cluster environments
  - Parallel efficiency drops rapidly as the number of processors increases
    - But: Relaxing barrier synchronization does not impact PH convergence
  - Bundling scenarios might help with parallel scaling
    - May increase number of iterations required
- PH can provide bounds!
  - Now comes with (rather tight) lower bounds
  - See “Obtaining Lower Bounds from the Progressive Hedging Algorithm for Stochastic Mixed-Integer Programs”



## 9. Solving Bilevel Problems



*Exceptional  
service  
in the  
national  
interest*



U.S. DEPARTMENT OF  
**ENERGY**



Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

# Overview of Bilevel Programming

---

A *bilevel program* is a mathematical program in which a subset of decision variables is constrained to take values associated with an optimal solution of a distinct, “lower” level mathematical program.

General formulation:

$$\begin{array}{ll} \min_{x \in X} & F(x, y) \\ \text{s.t.} & G(x, y) \leq 0 \\ & y \in P(x) \end{array}$$

} Upper-level problem

where

$$\begin{array}{ll} P(x) = & \operatorname{argmin}_{y \in Y} f(x, y) \\ & \text{s.t.} \quad g(x, y) \leq 0 \end{array}$$

} Lower-level problem

# Example: Modeling Security Problems

---



- Opponents must anticipate each other's moves
- Strategy should account for how opponent (best) responds

Extremely complex

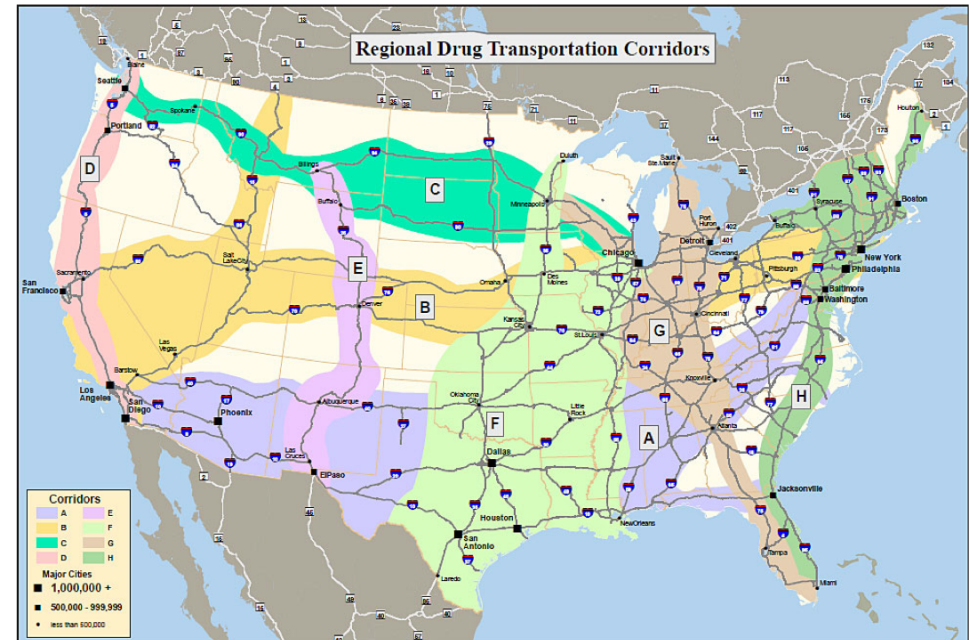
- Impossible to enumerate the set of all states in the game

Stackelberg games - bilevel programs

# Example: Smuggling Interdiction

Interdicator minimizes the potential for a smuggler to evade detection

- Interdicator installs defenses ( $x$ ) to minimize smuggler's evasion probability
- Smuggler traverses path ( $y$ ) that maximizes the probability of evasion



*Origin-destination nodes maybe unknown*

- $F(x,y)$  – evasion probability
- $X$  – interdicator's constraints (e.g. resource, budget)
- $P(x)$  – feasible paths given  $x$

# Modeling Bilevel Programs

---

No algebraic modeling language currently provides an *intuitive* syntax for expressing the structure of bilevel programs!

MPEC formulations are supported in several AMLs

- AMPL, AIMMS, GAMS, ...
- MacMPEC includes bilevel programs that are reformulated as single-level programs using optimality conditions

Bilevel problems can be expressed in several modeling languages:

- GAMS, YALMIP
- Explicitly pass variables and constraints to a bilevel solver

# A Simple Bilevel Example

---

Practical Bilevel Optimization: Algorithms and Applications

Jonathan Bard

Example 5.1.1

$$\begin{array}{ll} \min_{x \geq 0} & x - 4y \\ \text{s.t.} & \min_{y \geq 0} \quad y \\ & \text{s.t.} \quad -x - y \leq -3 \\ & \quad -2x + y \leq 0 \\ & \quad 2x + y \leq 12 \\ & \quad -3x + 2y \leq -4 \end{array}$$

# Modeling Example 5.1.1 with YALMIP

```
sdpvar x, y;
```

```
OO = x - 4y;
```

```
CO = [x] >= 0;
```

```
OI = y;
```

```
CI = [[y] >= 0,
 -x - y <= -3,
 -2x + y <= 0,
 2x + y <= 12,
 -3x + 2y <= -4];
```

```
solvebilevel(CO,OO,CI,OI,[y])
```

- Solve a bilevel problem using a simple branching strategy.
- Upper level problem defined by CO and OO
- Lower level problem defined by CI and OI, with decision variable y.

Note: This example adapted from the YALMIP bilevel documentation.



# Modeling Example 5.1.1 with GAMS

```
positive variables x,y; variables objout,objin;
equations defout,defin,e1,e2,e3,e4;
```

```
defout.. objout =e= x - 4*y;
defin.. objin =e= y;
```

```
e1.. - x - y =l= -3;
e2.. -2*x + y =l= 0;
e3.. 2*x + y =l= 12;
e4.. 3*x - 2*y =l= 4;
```

```
model bard / all /;
```

```
$echo bilevel x min objin * defin e1 e2 e3 e4 > "%emp.info%"
```

```
solve bard us emp min objout;
```

Writes an “empinfo” file that tells the solver that this is a bilevel problem with a lower level problem that minimizes objective *objin* with variables *y* subject to the constraints (*defin*), *e1*, *e2*, *e3* and *e4*.

# Modeling Example 5.1.1 with Pyomo

```
from pyomo.environ import *
from pyomo.bilevel import *
```

```
M = ConcreteModel()
M.x = Var(bounds=(0, None))
M.y = Var(bounds=(0, None))
M.o = Objective(expr=M.x - 4*M.y)
```

```
M.sub = SubModel(fixed=M.x)
M.sub.o = Objective(expr=M.y)
M.sub.c1 = Constraint(expr=- M.x - M.y <= -3)
M.sub.c2 = Constraint(expr=-2*M.x + M.y <= 0)
M.sub.c3 = Constraint(expr= 2*M.x + M.y <= 12)
M.sub.c4 = Constraint(expr=-3*M.x + 2*M.y <= -4)
```

- Lower level problem is declared with a *SubModel* component.
- The *var* argument indicates the lower level variables.
- Objectives, variables and constraints for the lower level problem are declared within this component.

# Pyomo Extensions for Bilevel Programs

---

## Modeling extensions

- Modeling components (`pyomo.bilevel`)

## Model transformations

- Can be applied automatically

## *Custom solvers*

- *Solvers tailored for specific classes of bilevel problems*

# Solving Bilevel Problems

---

Goal: Enable solution of bilevel problems with standard solvers

Process:

- Model problem with SubModel components
- Transform the problem to a standard form
  - LP, MIP, etc
- Apply a suitable solver

Reformulations for linear bilevel programming (BLP)

- A. BLP with continuous variables
- B. Quadratic minimax with continuous lower-level variables

# (A) BLP with Continuous Variables

---

Problem:

$$\begin{aligned}
 \min_{x \geq 0} \quad & c_1^T x + d_1^T y \\
 \text{s.t.} \quad & A_1 x + B_1 y \leq b_1 \\
 \min_{y \geq 0} \quad & c_2^T x + d_2^T y \\
 & A_2 x + B_2 y \leq b_2
 \end{aligned}$$

Reformulation: Replace lower-level problem with corresponding optimality conditions

$$\begin{aligned}
 \min \quad & c_1^T x + d_1^T y \\
 \text{s.t.} \quad & A_1 x + B_1 y \leq b_1 \\
 & d_2 + B_2^T u - v = 0 \\
 & b_2 - A_2 x - B_2 y \geq 0 \perp u \geq 0 \\
 & y \geq 0 \perp v \geq 0 \\
 & x \geq 0, y \geq 0
 \end{aligned}$$

# (A) BLP with Continuous Variables (cont'd)

---

Idea: Analyze the MPEC reformulation

Example:

- Use a custom solver that considers complementarity conditions (Bard, 1998)

Example:

- Chain reformulations: BLP  $\rightarrow$  MPEC  $\rightarrow$  GDP  $\rightarrow$  MIP
- Provide “BigM” values for unbounded variables
- Apply standard MIP solver (Fortuny-Amat and McCarl, 1981)

Example:

- Reformulate the complementarity conditions with nonlinear constraints (Ferris and Dirkse, 2005)

## (B) Quadratic Min/Max

---

Problem:

- Upper level constraints do not constrain  $y$

$$X = \{x | A_1 x \leq b_1, x \geq 0\}$$

- The upper decision variables may binary

$$\begin{aligned} \min_{x \in X} \quad & \max_{y \geq 0} \quad c_1^T x + d_1^T y + x^T Q y \\ \text{s.t.} \quad & A_2 x + B_2 y \leq b_2 \end{aligned}$$

Reformulation: Replace lower-level problem with the linear dual

$$\begin{aligned} \min \quad & c_1^T x + (b_2 - A_2 x)^T v \\ \text{s.t.} \quad & B_2^T v \geq d_1 + Q^T x \\ & A_1 x \leq b_1 \\ & x \geq 0, v \geq 0 \end{aligned}$$

## (B) Quadratic Min/Max

(cont'd)

Case 1:

- $A_2 \equiv 0$
- The reformulation is a simple LP (or a MIP if  $x$  are binary)

Case 2:

- The upper-level decision variables  $x$  are binary
- Reformulate the bilinear objective terms as disjunctions:

$$\begin{aligned}
 \min \quad & c_1^T x + b_2^T v - 1^T z \\
 \text{s.t.} \quad & B_2^T v \geq d_1 + Q^T x \\
 & x_i = 0 \quad \wedge \quad x_i = 1 \\
 & z_i = 0 \quad \wedge \quad z_i = A_2^T(i,*)v \\
 & x \in X, v \geq 0
 \end{aligned}$$

- Reformulate this GDP -> MIP using “BigM” values



# Solving Bilevel Programs in Pyomo

---

## Python Script:

- Formulate the model
  - Apply desired model reformulations
  - Apply a suitable optimizer
- OR
- Directly analyze the model within Python
    - (e.g. using Pyomo's algebraic structure)

## Pyomo Command:

- Execute a command that executes a Pyomo meta-solvers
  - Performs suitable reformulations
  - Applies a suitable optimizer
  - Maps the solution to the original problem

```
pyomo solve --solver=bilevel_ld model.py
```

# A Scripting Example: Quadratic Min/Max

---

Problem:

$$\begin{array}{ll}\max_z & \max_{x_i} \quad z(6x_1 + 4x_2 + 2x_3) \\ & \text{s.t.} \quad 4x_1 + 2x_2 + x_3 \geq 5 \\ & \quad \quad x_1 + x_2 \geq 3 \\ & \quad \quad x_2 + x_3 \geq 4 \\ & \quad \quad z \leq 0, x_i \geq 0\end{array}$$

# Bilevel Pyomo Model

---

```
from pyomo.environ import *
from pyomo.bilevel import *

M = ConcreteModel()
M.z = Var(within=NonPositiveReals)
M.x1 = Var(within=NonNegativeReals)
M.x2 = Var(within=NonNegativeReals)
M.x3 = Var(within=NonNegativeReals)
M.o = Objective(expr=M.z*(6*M.x1 + 4*M.x2 + 2*M.x3), \
 sense=maximize)

M.sub = SubModel(fixed=M.z)
M.sub.o = Objective(expr=M.o.expr, sense=minimize)
M.sub.c1 = Constraint(expr=4*M.x1 + 2*M.x2 + M.x3 >= 5)
M.sub.c2 = Constraint(expr=M.x1 + M.x2 >= 3)
M.sub.c3 = Constraint(expr=M.x2 + M.x3 >= 4)
```

# Script (Draft)

---

```
from pyomo.environ import *
from pyomo.bilevel import *
from problem import M

Fix upper-level variables
M.z.fixed = True

Dualize the submodel and set it on the model
xfrm = TransformationFactory('core.linear_dual')
M.dual = xfrm.create_using(M, options={'block':M.sub})

Unfix the upper-level variables
M.z.fixed = False
Deactivate the original objective and submodel
M.o.deactivate()
M.sub.deactivate()
```

# Pyomo Capabilities

---

## Relevant Pyomo Transformations

- `core.linear_dual`
- `bilevel.linear_dual`
- `bilevel.linear_mpec`
- `gdp.bigm`
- `gdp.bilinear`
- `gdp.chull`
- `mpec.simple_disjunction`
- `mpec.simple_nonlinear`

## Relevant Pyomo Meta-Solvers

- `bilevel_ld`

# Ongoing Work ...

---

- Generalization and maturation of transformations
  - E.g. Working with general BLP models
- Automatic recognition of bilevel structure
  - Can we automate the application of reformulations?
- Parameterizing transformations
  - How can we flexibly specify transformation options?
  - E.g. Specifying big-M values for specific complementarity conditions
- Additional meta-solvers
  - E.g. `bilevel_blp`

# Acknowledgements

---

- Richard L. Chen
- William E. Hart
- John D. Sirola
- Jean-Paul Watson