Optimization Software: Dakota and Pyomo

Julie Berge Ims

Haakon Eng Holck

Fall 2017

Advanced Process Simulation

Department of Chemical Engineering

Norwegian University of Science and Technology

Contents

1 Introduction **2**

2 Motivation **3**

2.1 Modelling . . . 3

2.2 Optimization . . . 3

3 Technology **5**

3.1 Pyomo . . . 5

3.1.1 Mathematical Model Design . . . 7

3.2 PySP . . . 8

3.2.1 Data Files . . . 11

3.3 Dakota . . . 11

3.3.1 Solvers . . . 12

3.3.2 An Illustration of Convergence Times . . . 14

3.3.3 The Dakota/Model Interface . . . 15

3.3.4 The Structure of the Dakota Input File . . . 16

4 Example Problems **17**

4.1 Maximum Flow Problem – LP . . . 17

4.2 Rosenbrock Function – NLP . . . 21

4.3 Birge and Louveaux's Farmer Problem – SP . . . 22

5 Discussion and Recommendations **27**

A Download and Installation of Software **31**

A.1 Pyomo 31

A.2 Dakota 32

B Pyomo code **33**

B.1 Command Line Arguments 33

B.2 Rosebrock Function 34

B.3 Maximum flow problem 37

C PySP code **42**

C.1 Command Line Arguments 42

C.2 Farmer problem 42

D Dakota code **51**

D.1 Dakota 51

D.2 Rosenbrock Function in Python 51

D.3 Rosenbrock Function in MATLAB 55

Chapter 1

Introduction

Optimization has been a fundamental concept in human history long before mathematical models and computers were developed. For instance, finding the best path down a mountain can be to follow the river stream. The river is a simulation of an old but gold solution for finding the optimal path. There is no single "best" solution method for optimization problems. Depending on the objective and the limitations, different methods are prefered. Today, can we use mathematical models to formulate such real-world phenomena in order to try to find the optimal path. Optimization softwares today try utilize these models to simulate and generate an optimal solution, subject to some constraints. This makes optimization software a useful tool. There are many applications and software libraries with implementations of optimization methods, some of which have a wide range of problem applications and some of which are specialized in one kind of optimization problem.

This report will examine two softwares used for optimization purposes, The Python Optimization Modeling Objects (Pyomo) and Dakota. These libraries are made for mathematical modelling and optimization, and they contain methods to solve most kinds of optimization problems. The technology behind the softwares will be explained as well as the construction and implementation of mathematical models. Examples of optimization problems will be investigated to better grasp the design and functionality of the two softwares. Dakota and Pyomo will be compared in order to make some recommendations for future applications. Code and installation guide is provided in the end.

Chapter 2

Motivation

2.1 Modelling

Modelling is an essential part in many aspects of engineering and scientific research. Modeling deals with how real-world phenomena can be formulated and simplified through mathematical equations. This is a way of structuring knowledge about real-world objects that can facilitate the analysis of the model. The structure also provides information about the communication of the knowledge about a mathematical model[1]. Models can be represented with a mathematical syntax and language in order to be analyzed using computational algorithms and mathematical theory.

2.2 Optimization

In principle, optimization consists of finding a minimum or maximum of a particular objective function, often while the input values are constrained within some domain. Optimization of mathematical models will utilize functions that represent objectives and constraints in order to design the system problem[1]. A generalized representation of an optimization problem can be the following:

$$\begin{aligned} \min \quad & (\phi(x,y)) \\ \text{s.t.} \quad & x \in X \\ & y \in Y \\ & f(x,y) \le \epsilon \end{aligned} \tag{2.1}$$

where $\phi(x, y)$ is the objective function which is to be minimized/maximized. x and y are variables and states, respectively. $f(x, y) \le \epsilon$ denotes the constraints. Note that the objective function is minimized using optimization algorithms. When $\phi(x, y)$ is to be maximized, the objective function is simply reformulated to $-\phi(x, y)$.

Chapter 3

Technology

3.1 Pyomo

The Python Optimization Modeling Objects (Pyomo) software package is an open-source software package for formulating and analysis of mathematical models for optimization applications. Pyomo is an Algebraic Modeling Language (AML), which is a high level programming language used to construct and solve mathematical problems, optimization problems in particular. The clean syntax of high level languages is more similar to how humans would develop and understand the structure of mathematical problems in comparison to low-level languages. AML software provides interfaces to external solver packages which is used to solve and analyze optimization problems[1].

Solver algorithms are often written in low-level language which offer sufficient performance to solve large numerical problems. However, low-level languages have complex syntax which makes it challenging to develop various applications. Pyomo has a new approach where the main objective is to provide a software package that enables the user to construct mathematical problems in a high-level language, Python, as well as providing interfaces for solvers for optimization problems. Python also supports objective oriented programming which allows the user to formulate both simple and complex mathematical problems in a concise manner. In this way Pyomo make use of the the flexibility and the supporting applications of the high-level language and the performance of the low-level language for numerical computations[1]. Pyomo

supports a wide range of problems and some of them are[2]:

- Linear programming (LP)
- Quadratic programming (QP)
- Nonlinear programming (NLP)
- Mixed-integer linear programming (MILP)
- Mixed-integer quadratic programming (MIQP)
- Mixed-integer nonlinear programming (MINLP)
- Stochastic programming (SP)
- Generalized disjunctive programming (GDP)
- Differential algebraic equations (DAE)
- Bilevel programming (BP)
- Mathematical programs with equilibrium constraints

Pyomo is open source which means that the source code for the particular software is available to use or modify as a developer or user. Performance limitations and other bugs can be identified and solved by computer nerds world wide and the reliability of the software can be improved. Pyomo was previously called Coopr and is now a software package contained in Common Optimization Python Repository (Coopr)[1]. Pyomo supports many of solvers, both commercial and open source and some of them are listed below[1]:

GLPK, IPOPT and CBC are open-source solvers. CPLEX and Gurobi are commercial solvers.

Table 3.1: Solvers supported by Pyomo[3]

	Mathematical problem						
Solver	LP	QP	NLP	MILP	MIQP	MINLP	SP
GLPK	x			x			x
IPOPT		x	x		x	x	
CPLEX	x	x		x	x		x
Gurobi	x	x		x	x		x
CBC	x	x		x	x		

3.1.1 Mathematical Model Design

Pyomo supports abstract and concrete model designs. Abstract models separates the declaration from the data that generates a model instance. Data may be stored in an external database or spreadsheet. These data files would typically contain values for different variables declared in the model file. Abstract models are useful for more complex problems that require a large number of data. Concrete models will have the data and the model instance together. The examples later on in the report will illustrate the design for abstract and concrete models. However, there are some basic steps for the modelling process for both abstract and concrete models[1]:

1. Create model and declare components
2. Instantiate the model
3. Apply solver
4. Interrogate solver results

A mathematical model in Pyomo requires certain components that define different aspects of the model. [1]The components are defined in Pyomo through Python classes:

- `Set` set data that is used to define a model instance
- `Param` parameter data that is used to define a model instance
- `Var` decision variables in a model
- `Objective` expressions that are minimized or maximized in a model

- `Constraint` constraint expressions in a model

The Maximum flow problem is a mathematical optimization problem and the implementation of the model is discussed later in the report.

3.2 PySP

It is evident that data may be uncertain in the real world. Stochastic programming is a mathematical framework used to express and solve problems where parameter uncertainty is independent of decisions and that these parameters may be identified on a later stage[4]. PySP is an extension to Pyomo which provides generic inferfaces for solvers for stochastic programming[1]. This means that the original mathematical model design for pyomo is also available in PySP.

Modelling packages that aim to solve stochastic problems will require a translation of the problem into its extensive form[1]. Extensive form is a deterministic mathematical programming formulation of the stochastic problem where all possible uncertain parameter outcomes are represented and solved simultaneously. A deterministic formulation is when the parameter values fully define the output. Standard solvers are able to solve and optimize this extensive form of the stochastic program. By utilizing the high-level programming language (Python) and the incorporation of the mathematical programming formulation in that language (Pyomo), PySP provides highly configurable and completely generic solver implementations for stochastic programs[1].

Stochastic optimization problems will contain uncertain parameters and corresponding probability distributions. These probability distributions can be converted into discrete values by having a finite number of outcomes for each variable, high, medium or low for instance. A scenario tree is then generated as a way to structure all the possible combinations of parameter realizations. Each node will be a parameter realization for a particular random variable. A scenario is a combination of random parameter realization with associated probability of occurrence and will act as a path from the root node of the tree to one of the leaves[1]. The illustration below indicates how a scenario tree is constructed with only one uncertain parameter.

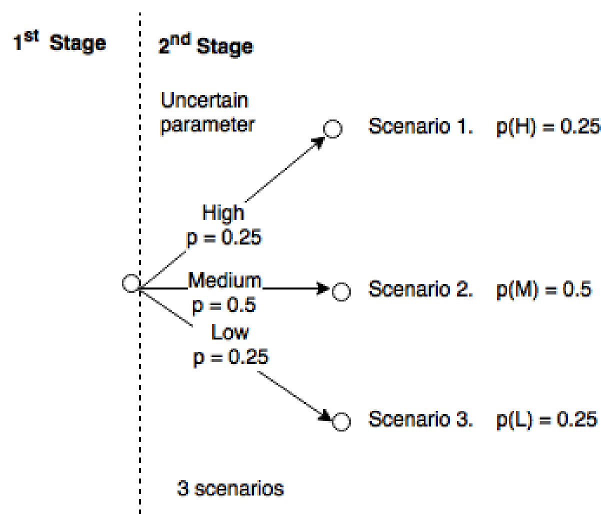

Figure 3.1: Illustration of scenario tree with two stages and one uncertain parameter which is results in three scenarios

The first stage is an initial stage before accessing any random variable parameter realization. The second stage introduces new information which in this case is the three possible outcomes for the random variable[4]. Since this problem only has one uncertain parameter, each of the possible outcomes represent a scenario. Some variables are introduced to illustrate how the stochastic problem is translated into its extensive form:

- x regular parameters and decision variables
- $f(x)$ function which is not affected by the uncertain parameter outcome
- z uncertain parameter
- $g(x,z)$ function which is affected by the uncertain parameter outcome

This means that an objective function could be translated into the following expression:

$$\text{Objective func.} = \underbrace{f(x) + g(x,z)}_{\text{original formulation}} = \underbrace{f(x) + 0.25\ g(x,z_H) + 0.5\ g(x,z_M) + 0.25\ g(x,z_L)}_{\text{extensive formulation}}$$

Here the different scenarios are presented and solved simultaneously. Standard solvers are able to solve and optimize this deterministic formulation of the stochastic program. Constraints will

also be affected by the uncertainty, however, an example will presented later to illustrate this.

The next illustration is an example of a scenario tree with four stages and three uncertain parameters. Each scenario is a complete set of random variable realizations and an associated probability of occurrence.

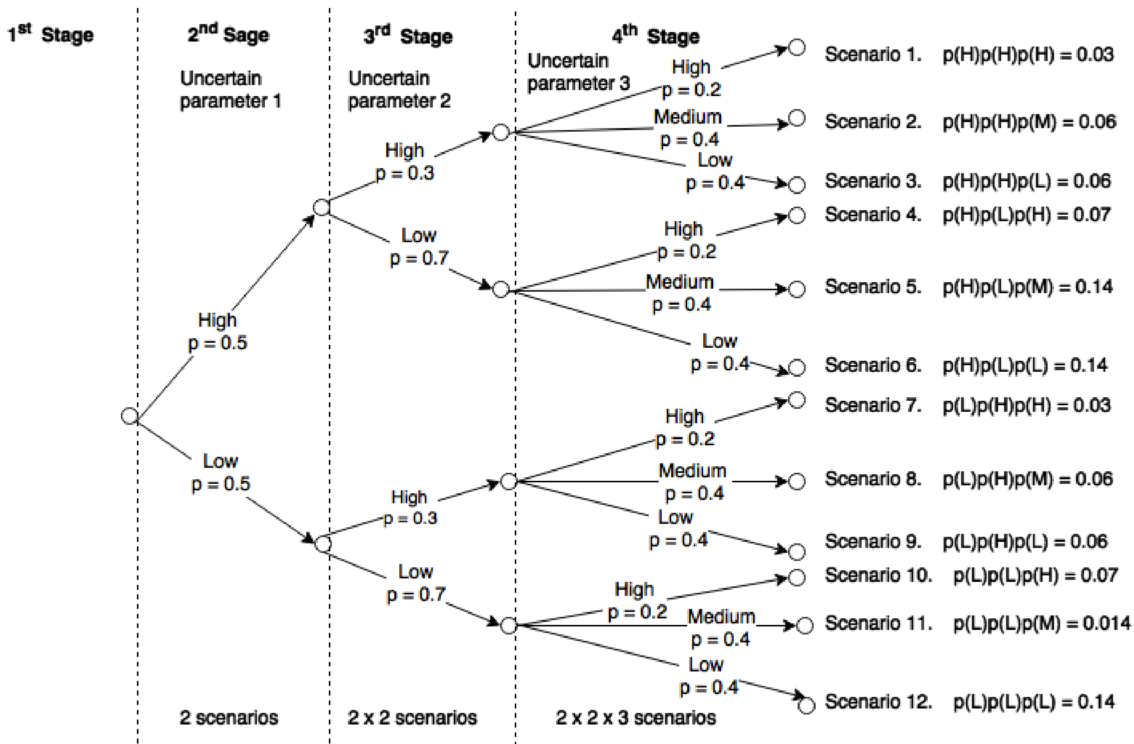

Figure 3.2: Illustration of scenario tree with four stages and three uncertain parameters which is results in 12 scenarios

For simplicity, lets assume that each stage after the initialization in the first stage corresponds to a new random variable. It is possible to have two or more random variables in one stage, but we will consider only one per stage. That means that illustration is a four stage model with three random variables. In last three stages, new information is revealed that was not available in previous stages. This is how nonanticipativity is enforced, random variable outcomes are not affected by information about the past to make future decisions. The deterministic model is generated through this scenario tree[1]. The random variables in the different scenarios can also indicate the development of the uncertain parameters over time.

3.2.1 Data Files

PySP require that the data files for the stochastic optimization contain particular code and have specific names[1]. The data files listed below are required by PySP in order to compile and run properly.

- `ReferenceModel.py` deterministic reference Pyomo model
- `ReferenceModel.dat` reference model data
- `ScenarioStructure.py` PySP model to specify the structure of the scenario tree
- `ScenarioStructure.dat` data to instantiate the parameters/sets of the scenario tree
- `AboveAverageScenario.dat` data for above average scenario
- `AvearageScenario.dat` data for average scenario
- `BelowAverageScenario.dat` data for below average scenario

The names of the last three data files are not specific to `AboveAverageScenario.dat`, `AverageScenario.dat` or `BelowAverageScenario.dat`. However, you must call the data files the same as you call each scenario in `ScenarioStructure.dat`. Examples of these data files are attached in Appendix.

3.3 Dakota

Dakota is an open-source simulation and optimization software developed by Sandia National Laboratories. Its purpose is to *"provide engineers and other disciplinary scientists with a systematic and rapid means to obtain improved or optimal designs or understand sensitivity or uncertainty using simulation based models."*[5] Dakota was originally meant to interface simulation data with an archive of optimization algorithms (for use in structural analysis and design problems), but its capabilities has been expanded to include other kinds of analyses:

- **Parameter studies:** Exploring the effects of changes to the parameters of a simulation model. This can provide information on smoothness, multi-modality, robustness and nonlinearity of the simulation.

- **Design of experiments:** Design and analysis of computer experiments (DACE) methods. Meant to provide a good range for input parameters for experiments.
- **Uncertainty quantification:** Calculating the uncertainty propagation from inputs to outputs.
- **Optimization:** Optimization of a simulation model. Different optimization algorithms are available: Gradient based and derivative free, local and global optimization algorithms. Multi-objective optimization and optimization of surrogate models is also available.
- **Calibration:** Estimating the parameters of a simulation model to fit an experimental data set (also known as the inverse problem). This is simply a twist on regular optimization problems, as it minimizes the squared error between simulation data and experimental data as a function of the model parameters. Dakota uses specialized optimization methods that are well suited to exploit the structure of these kinds of problems

These are all iterative analysis methods, which work by treating the simulation model like a black box; Dakota knows nothing about the model apart from its outputs as a response to inputs (an illustration is shown in figure 3.3). The algorithms implemented in Dakota are also designed to allow for parallel computing.

3.3.1 Solvers

Dakota has a collection of different optimization solvers:

- **Local gradient-based methods:** These methods are the best methods to use on a smooth, unimodal (only one minimum) system. The gradients (and preferably, hessian matrixes) can be provided as an output from the system (analytic gradients), or be estimated using a finite difference method. Analytic gradients are faster and more reliable, as the finite difference method requires additional function evaluations and only returns an estimated gradient.
- **Local derivative-free methods:** These methods are (as the name suggests) not reliant on gradients, and are therefore more robust than the gradient-based methods. These can

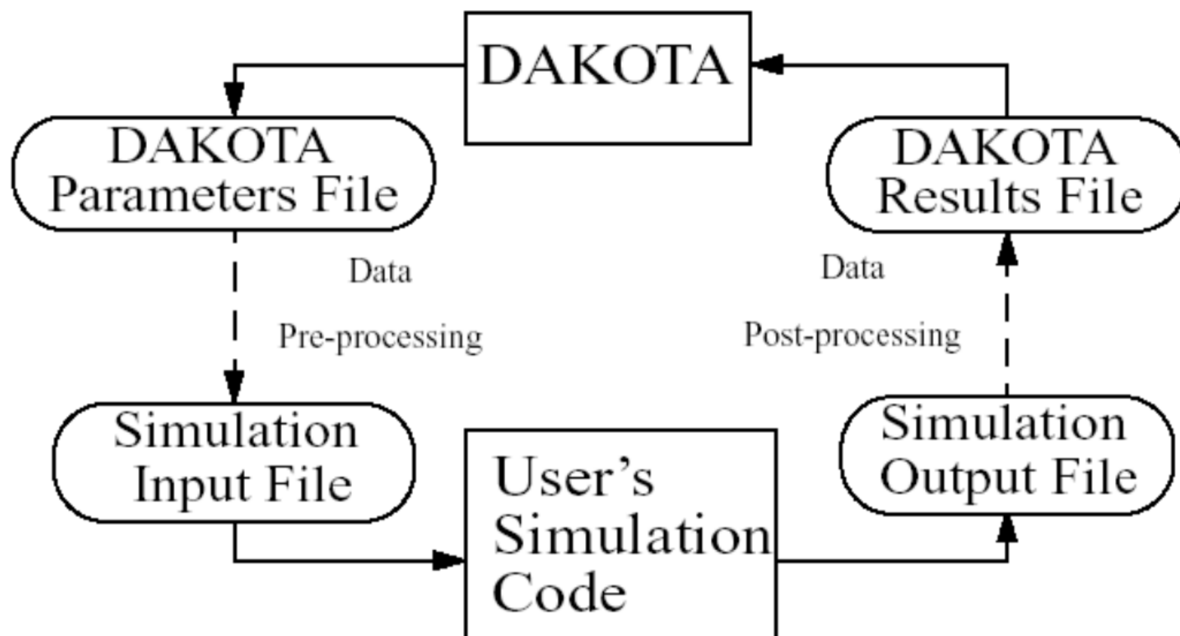

Figure 3.3: An illustration of the interface between Dakota and the simulation model. The figure is taken from the user manual.[5]

be used on nonsmooth and "poorly behaved" problems. The convergence rates for the derivative-free methods are slower than the gradient-based methods, however. Nonlinear constraints are also a bigger difficulty in derivative-free methods; the best way to handle nonlinear constraints is still an area of research.

- **Global methods:** Global methods are (excepting multistart methods) derivative-free. They contain stochastic elements, which means that the program will behave differently (and may produce different results) if you run it multiple times. Global methods can provide good (but not necessarily optimal) answers on noisy, multimodal problems. As with the local derivative free methods, nonlinear constraints makes the problem more complex.

In each of these cases, there are different solvers that are more fitting for some problem types (constrained/unconstrained, with and without equality constraints and so on). Most of the solvers are third-party solvers, but the majority of these are still included in the Dakota installation.

3.3.2 An Illustration of Convergence Times

In figure 3.4, the convergence rate for a set of different solvers is shown. All methods were used on the Rosenbrock function (see section 4.2) with initial values $x1 = 1,2$, $x2 = 2$.

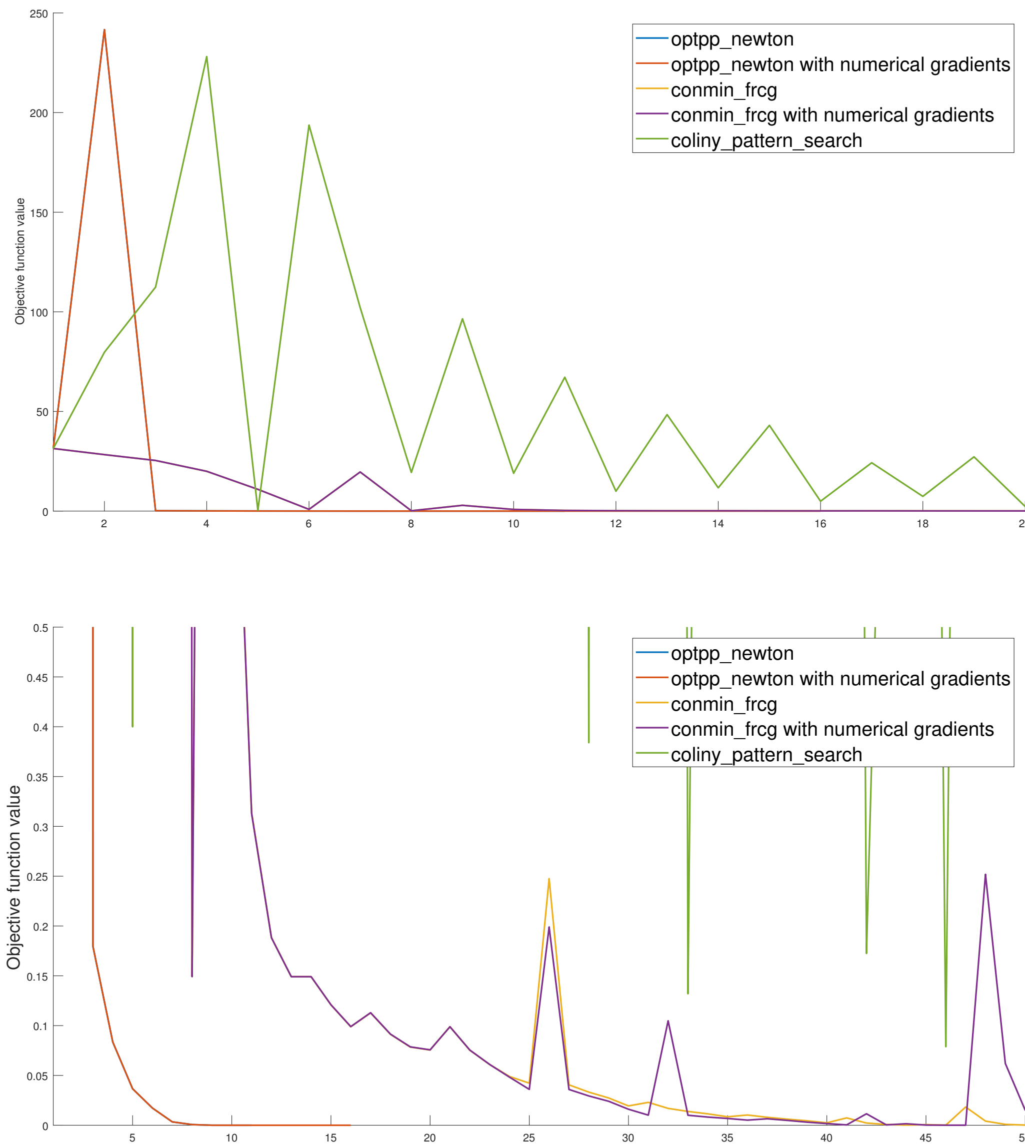

Figure 3.4: The objective function value as a function of the iterations for a collection of optimization methods. Both plots are of the same values, with different scales on the axes (the lower plot has a smaller scale in the y-axis but a larger scale on the x-axis). In the cases where the same solver is used with and without numerical gradients, the function values are almost identical and only one of the lines are visible.

The fastest to converge is the `optpp_newton` method, which uses both gradients and hessians.

The conmin_frcg method (a *conjugate gradient* method), which only uses gradients, converged slower. The derivative-free *pattern search* method, coliny_pattern_search, timed out after reaching the maximum amount of iterations. It was slowly trending towards the optimum, but would require a (comparatively) huge amount of time to converge.

The iterations shown in figure 3.4 does not tell the whole story, however. The methods that estimated the gradients numerically needed additional computation, which is not reflected in the iterations that are counted in the data files. The runtime for each method is given in table 3.2.

Table 3.2: The runtimes for the different methods.

Method	Runtime (s)
optpp_newton	2.259
optpp_newton with numerical gradients	7.994
conmin_frcg	10.263
conmin_frcg with numerical gradients	18.406
coliny_pattern_search	N/A

3.3.3 The Dakota/Model Interface

As shown in figure 3.3, some work is required to bridge the gap in the interface between the simulation model and Dakota. Dakota reads the model outputs from a results file, and writes the inputs into a parameter file. This means, conversely, that the interface needs to be constructed to read the input values from the parameter file and write the outputs into the results file (pre-processing and post-processing respectively). The default Dakota installation does not have this functionality integrated or automated in any way, which means that there is some work required in writing the wrapper function (the function that "wraps" around the actual model function, handling the interfacing with Dakota) for any given problem. On the plus side, this wrapper (as well as the simulation model) can be written in any kind of programming language that is able to read and write to the parameter- and results file. This makes Dakota flexible and minimizes the work required to make the model compatible with the optimization/analysis program.

If installed manually from source files, Dakota can be configured to interface directly with Python, Matlab or Scilab. This removes the need for the wrapper function.

3.3.4 The Structure of the Dakota Input File

An example of an input file is shown in appendix D.2. It shows the six "specification blocks": `environment`, `method`, `model`, `variables`, `interface` and `responses`.

- In the `environment` block, you can specify if you want graphical- or tabular data output and some other advanced options for some methods. This block is optional, as these output files are not required.
- The `method` block contains the information about the analysis method you want to use on the simulation model. Here you specify what method you want to use, as well as the options associated with the method. For advanced studies, multiple method blocks may be used in one input file.
- The `model` block tells Dakota about what to expect from the simulation model. In simple models, this block is unnecessary, as it will default to the `single` option, which means that Dakota will expect a single set of variables and responses, as well as an interface.
- The variables of the model are specified in the `variables` block, where the type (for example `continuous_design/discrete_design_range/normal_uncertain` or so on) is specified, as well as other relevant options. For continuous variables, the options would be things like initial values (although no information is strictly required for `continuous_design`) while for normally distributed variables, the mean and standard deviation must be specified.
- The `interface` block is where Dakota is linked to the specific simulation model. The `fork` keyword tells Dakota that the model considered is external from Dakota, which means it needs to create a results- and a parameters-file. It also calls the wrapper function specified in the `analysis_driver`.
- In the `responses` block, Dakota is told what it can expect as the outputs from the simulation model. The `response_function` specifies the amount of responses, while keywords like `no_gradients` vs. `analytic_gradients` vs. `numerical_gradients` specify whether the model will calculate and return the gradients or not.

Chapter 4

Example Problems

Three different optimization problems are introduced, and the problem formulation and solution method (in Pyomo) is shown. The maximum flow problem is an example of linear programming, the Rosenbrock function is a non-linear program and the Birge and Louveaux's farmer problem is chosen to demonstrate the functionality of PySP on a stochastic optimization problem. Dakota will also be used to minimize the Rosenbrock function.

4.1 Maximum Flow Problem – LP

Maximum flow problems are used to find feasible flow through a single-source, single-sink flow network. The problem consists of nodes and edges which are linking the nodes together. The edges have capacity and direction. The capacity is the maximum flow through a particular edge. The objective is to maximize the total flow into the sink from the source[6]:

$$\max\Big(\sum(\text{Flow into Sink}\Big) \tag{4.1}$$

This means that the capacity of the edges into the sink is maximized, but the flow out of the source is not necessary at its maximum. The combination of flow through the edges between the source and the sink will yield the optimal solution.

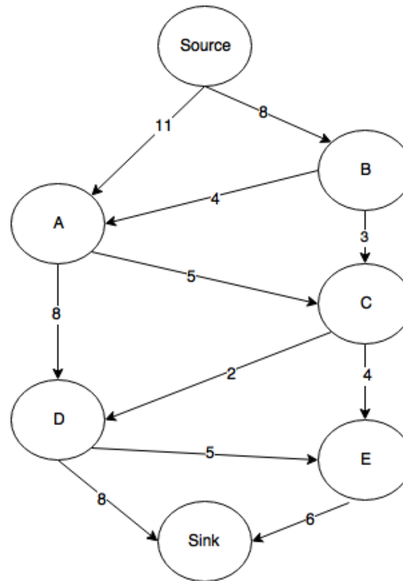

Figure 4.1: Illustration of a network with source ans sink . The numbers denote capacity of the flow and the arrows denote direction of the flow.

The objective is subject to two constraints[6]:

- **Capacity constraints:**

  The flow f through an edge e cannot exceed its capacity c

  $\forall\ e: \quad f(e) \leq c(e)$

- **Flow conservation:**

  Total flow into a node n is equal to the flow flowing out of the node. For all nodes except the Source and the Sink

  $\forall\ \ n(n \neq \text{ Source and } n \neq \text{ Sink}): \ \sum f_{\text{into n}} - \sum f_{\text{out of n}} \ = 0$

The illustration above is the numerical example that is coded in the end of this report. The problem can be formulated into a linear optimization problem (LP). $f_{i,j}$ denotes flow from node i to node j. The objective is formulated as:

$$\max\left(\sum (f_{D,\text{Sink}} + f_{E,\text{Sink}}\right) \tag{4.2}$$

The objective is subject to constraints, flow capacity and flow conservation constraints. Flow capacity constraints are defined as:

$$f_{\text{Source},B} \leq 8 \tag{4.3}$$

$$f_{\text{Source},A} \leq 11 \tag{4.4}$$

$$f_{B,A} \leq 4 \tag{4.5}$$

$$f_{A,C} \leq 5 \tag{4.6}$$

$$f_{B,C} \leq 3 \tag{4.7}$$

$$f_{A,D} \leq 8 \tag{4.8}$$

$$f_{C,D} \leq 2 \tag{4.9}$$

$$f_{C,E} \leq 4 \tag{4.10}$$

$$f_{D,E} \leq 5 \tag{4.11}$$

$$f_{E,\text{Sink}} \leq 6 \tag{4.12}$$

$$f_{D,\text{Sink}} \leq 8 \tag{4.13}$$

$$\tag{4.14}$$

Flow conservation constraints are defined as:

$$\text{Node A:} \quad f_{A,C} + f_{A,D} - f_{\text{Source},A} - f_{B,A} = 0 \tag{4.15}$$

$$\text{Node B:} \quad f_{\text{Source},B} - f_{B,A} - f_{B,C} = 0 \tag{4.16}$$

$$\text{Node C:} \quad f_{A,C} + f_{B,C} - f_{C,D} - f_{C,E} = 0 \tag{4.17}$$

$$\text{Node D:} \quad f_{A,D} + f_{C,D} - f_{D,E} - f_{D,\text{Sink}} = 0 \tag{4.18}$$

$$\text{Node E:} \quad f_{C,E} + f_{D,E} - f_{E,\text{Sink}} = 0 \tag{4.19}$$

$$\tag{4.20}$$

The model data is in the file `maxflow.dat` and the mathematical problem is formulated in code in the file `maxflow.py`. The implementation of `maxflow.py` requires certain components that define different aspects of the model, `Set, Param, Var, Objective` and `Constraint`. This

first line of code is necessary to create a model object in the `maxflow.py` file.

```
# Creating the model object
model = AbstractModel()
```

The next lines are creating different sets, parameters and variables using suitable components such as `Set`, `Param` and `Var` to define the model.

```
# Nodes in the network
model.N = Set()
# Network arcs
model.A = Set(within=model.N*model.N)
# Source node
model.s = Param(within=model.N)
# Sink node
model.t = Param(within=model.N)
# Flow capacity limits
model.c = Param(model.A)
# The flow over each arc
model.f = Var(model.A, within=NonNegativeReals)
```

The next lines will construct the objective using the `Objective` component.

```
# Maximize the flow into the sink nodes
def total_rule(model):
    return sum(model.f[i,j] for (i, j) in model.A if j == value(model.t))
model.total = Objective(rule=total_rule, sense=maximize)
```

The very last section of the code will enforce constraints on the mathematical problem. The `Constraint` component is used to define an upper limit on the flow across each edge and enforce flow through each node.

```
# Enforce an upper limit on the flow across each arc
def limit_rule(model, i, j):
```

```
    return model.f[i,j] <= model.c[i, j]
model.limit = Constraint(model.A, rule=limit_rule)
# Enforce flow through each node
def flow_rule(model, k):
    if k == value(model.s) or k == value(model.t):
        return Constraint.Skip
    inFlow  = sum(model.f[i,j] for (i,j) in model.A if j == k)
    outFlow = sum(model.f[i,j] for (i,j) in model.A if i == k)
    return inFlow == outFlow
model.flow = Constraint(model.N, rule=flow_rule)
```

The same components are used to construct models in Pyomo and PySP in the next examples. This code make out the `maxflow.py` file and the corresponding data file and results are added later in the report.

4.2 Rosenbrock Function – NLP

The Rosenbrock function is a mathematical optimization problem often used for performance testing for optimization solvers and algorithms. This problem is also known as Rosenbrock's banana function or Rosenbrock's valley. The reason for this is because it is a non-convex function with a global minimum in a long, narrow, parabolic shaped valley. It is easy to find the valley, but to converge to the global minimum is rather difficult. The objective function is defined by[7]:

$$f(x,y) = (a-x)^2 + b(y-x^2)^2 \tag{4.21}$$

The main objective is to minimize this function an find the global minimum which is at $(x,y) = (a,a^2)$ where $f(x,y) = 0$. a and b are most often set to be 1 and 100 respectively. This is a non-linear problem, so non-linear solvers must be applied. The figure below is a plot of the Rosenbrock function of two variables. The same components used in the implementation of the previous Maximum flow problem are used to construct this mathematical optimization problem.

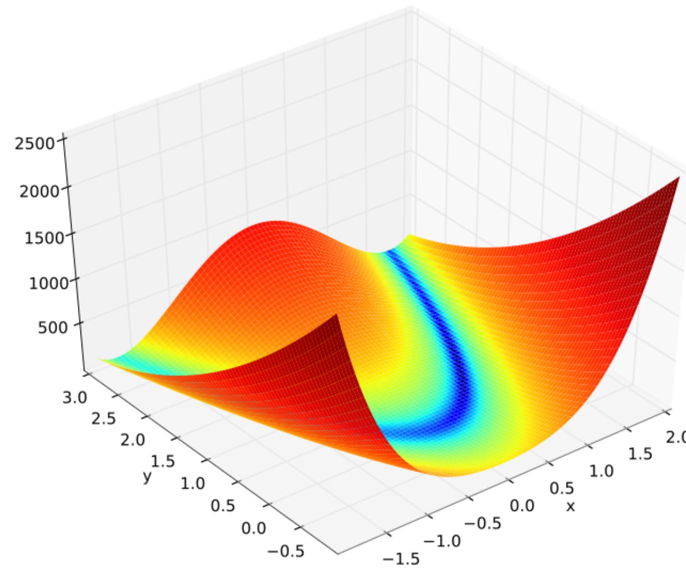

Figure 4.2: Plot of the Rosenbrock funciton were a = 1 and b= 100. The global miniumum is at (1,1)[7]

4.3 Birge and Louveaux's Farmer Problem – SP

Birge and Louveaux have created a problem to illustrate how one can create and solve a stochastic optimization problem. This stochastic optimization problem will be solved using PySP. Farmer Ted has 500 acres where he can grow wheat, corn, or beans. The cost per acre is $150, $230 and $260, respectively. Ted requires 200 tons of wheat and 240 tons of corn to feed his cattle. Production in excess of these amounts can be sold for $170/ton wheat and $150/ton corn. If any shortfall, additional products must be bought from the wholesaler at a cost of $238/ton wheat and $210/ton corn. Farmer Ted can also grow beans which are sold at $36/ton for the first 6000 tons. The remaining tons will only be sold at $10/ton beans, due to economic quotas on bean production. The average expected yield is 2.5, 3 and 20 tons per acre for wheat, corn and beans, respectively[8]. This information can be used to introduce necessary decision variables:

- x_w, x_c and x_b are acres of Wheat, Corn, Beans planted, respectively
- w_w, w_c and w_b are tons of Wheat, Corn, Beans sold, respectively (at favorable price)
- e_b is tons of beans sold at lower price
- y_w and y_c are tons of Wheat, Corn purchased

Decision variables are the variables which we can decide, and they can be parameters, vectors, a function etc. The main objective of this problem is to maximize the expected profit for Ted which is done by finding the *best* suitable values for the decision variables. Based on the information introduced earlier and the decision variables, the objective function for this problem can be defined as follows:

$$\begin{aligned}\text{Exp.profit} = -150x_w - 230x_c - 260x_b &\left.\right\}\textbf{Fixed}\text{ cost of total acres planted}\\ -238y_w + 170w_w &\left.\right\}\text{Total profit from production of wheat}\\ -210y_c + 150w_c &\left.\right\}\text{Total profit from production of corn}\\ +36w_b + 10e_b &\left.\right\}\text{Total profit from production of beans}\end{aligned}$$

The objective function is subject to constraints according to earlier information. The constraints can be defined as:

$$x_w + x_c + x_b \leq 500 \tag{4.22}$$

$$2.5x_w + y_w - w_w = 200 \tag{4.23}$$

$$3x_c + y_c - w_c = 240 \tag{4.24}$$

$$20x_b - w_b - e_b = 0 \tag{4.25}$$

$$w_b \leq 6000 \tag{4.26}$$

$$x_w, x_c, x_b, y_w, y_c, e_b, w_w, w_c, w_b \geq 0 \tag{4.27}$$

At this stage, this is a deterministic problem as we assume that all data are known. The deterministic problem is defined in `ReferenceModel.py`. PySP requires that the reference model is in a file named `ReferenceModel.py`. The the corresponding model data is in the file `ReferenceModel.dat`.

In equations 4.23-25, there are constants in front of the x-values, $2.5x_w$, $3x_c$ and $20x_b$. These constants denote yield of the particular substance planted. Yield is considered an uncertain parameter as it may vary through time. Hence, we consider the possibility that the yield per acre could be lower or higher than average. Lets assume equal probability for each case. That means

that there is is a probability of 1/3 that the the yields will be the average values that were provided (i.e., wheat 2.5; corn 3; and beans 20). It is equal probability that the yields will be lower than average (2, 2.4, 16) and that the yields will be higher than average (3, 3.6, 24). It is assumed that above average has 20% higher yield and below average has 20% lower yield than the average. Each of these cases make out a set of data which we call a scenario and together they give rise to a scenario tree. This scenario tree is rather simple as there is only one uncertain parameter. The scenario tree for this particular problem would look something like this:

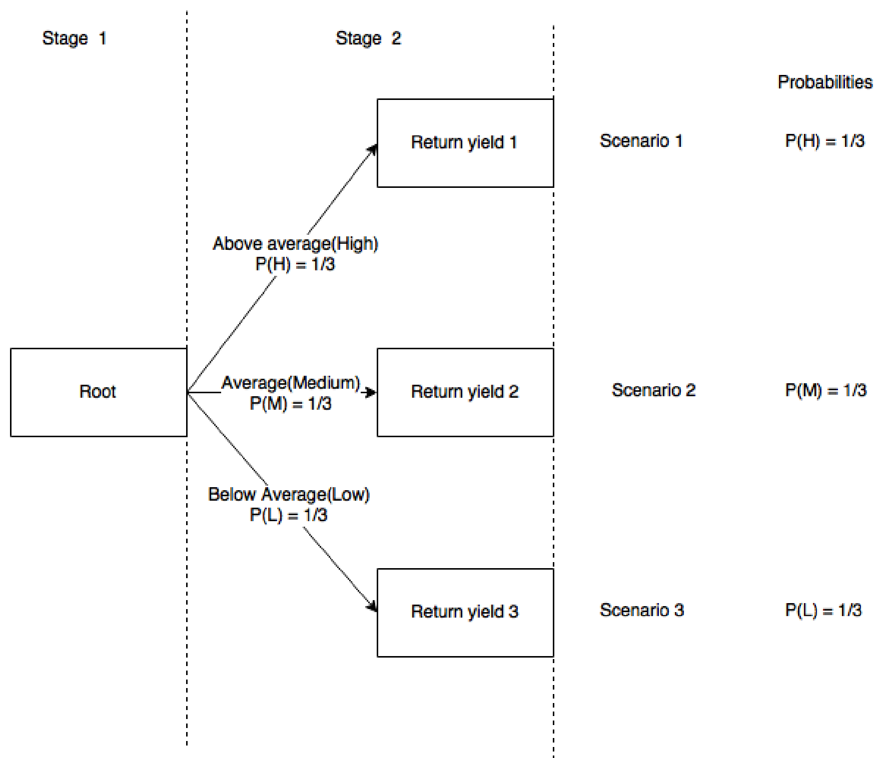

Figure 4.3: Caption

The root node with three leaf nodes which corresponds to each scenario. The cost of acre for each substance will not be affected by the yield. The other decision variables will depend on the scenario realization and are called second stage decisions. In order to make a systematic approach , attach a scenario subscript s = 1(above average), 2(average), 3(below average) to each

of the purchase and sale variables in to create a new objective function for the expected profit:

$$\begin{aligned}
&\text{Exp.profit} = -150x_w - 230x_c - 260x_b \\
&+ 1/3(-238y_{w1} + 170w_{w1} - 210y_{c1} + 150w_{c1} + 36w_{b1} + 10e_{b1}) \\
&+ 1/3(-238y_{w2} + 170w_{w2} - 210y_{c2} + 150w_{c2} + 36w_{b2} + 10e_{b2}) \\
&+ 1/3(-238y_{w3} + 170w_{w3} - 210y_{c3} + 150w_{c3} + 36w_{b3} + 10e_{b3})
\end{aligned}$$

The new constraints can then be defined as:

$$x_w + x_c + x_b \leq 500 \tag{4.28}$$

$$3x_w + y_{w1} - w_{w1} = 200 \tag{4.29}$$

$$2.5x_w + y_{w2} - w_{w2} = 200 \tag{4.30}$$

$$2x_w + y_{w3} - w_{w3} = 200 \tag{4.31}$$

$$3.6x_c + y_{c1} - w_{c1} = 240 \tag{4.32}$$

$$3x_c + y_{c2} - w_{c2} = 240 \tag{4.33}$$

$$2.4x_c + y_{c3} - w_{c3} = 240 \tag{4.34}$$

$$24x_b - w_{b1} - e_{b1} = 0 \tag{4.35}$$

$$20x_b - w_{b2} - e_{b2} = 0 \tag{4.36}$$

$$16x_b - w_{b3} - e_{b3} = 0 \tag{4.37}$$

$$w_{b1}, w_{b2}, w_{b3} \leq 6000 \tag{4.38}$$

$$\text{All variables} \geq 0 \tag{4.39}$$

The original stochastic problem is translated into this extensive form where all the different scenarios are represented and solved simultaneously.

Recall that data for each scenario is contained in particular `.dat` files. The data in the Scenario-based data files in this case are identical, except for the very last line where the yield is specified. Repetitive code is not convenient for more complex problems, and can it be simplified for that matter. Node-based data-files can be used instead of `AboveAverageScenario.dat`,

`AverageScenario.dat` and `BelowAverageScenario.dat`. A data file called `RootNode.dat` will replace `ReferenceModel.dat` except it will not contain the very last line with the yield specification. `BelowAverageNode.dat`, `AverageNode.dat`, and `AboveAverageNode.dat` will only contain the line which specifies the yield. The data file `ScenarioStructure.dat` must contain the following line `param ScenarioBasedData := False ;` , because this is set to true by default[1]. The same components used in the implementation of the Maximum flow problem are used to construct this mathematical optimization problem.

Chapter 5

Discussion and Recommendations

While Dakota and Pyomo both have capabilities for optimization, the areas of use are quite different. For simply solving optimization problems, Pyomo has a lower entry barrier, as well as available solvers for multiple complex problem types, some of which are not included in Dakota. Dakota includes no methods for the stochastic programming discussed in this report, nor any LP solvers like the one used for the maximum flow problem. The lack of any simple LP solvers (like the easily implemented Simplex method) indicates that these kinds of methods are not typically applied on the types of models that Dakota is used for.

Pyomo is an excellent tool for developing a mathematical model from scratch and applying suitable solvers. The process of designing the mathematical model is quite straightforward as specific model components are defined in Pyomo through Python classes. These same components are used in PySP as this is an extension of Pyomo.

In comparison to Pyomo, Dakota requires more work to start using, as input files and wrapper functions have to be made before doing the analysis on the mathematical model. Therefore, it is a waste of time to use Dakota for solving an optimization problem that can quickly be set up and solved in Pyomo, MATLAB or similar. It is designed to be used as an auxiliary tool to evaluate a model that is already implemented. Dakota's interfacing means that no changes is needed on the model itself when performing an analysis. However, this interfacing adds quite a bit of computational overhead which affects the runtime. This can be seen when comparing the

times in table 3.2 (for Dakota) with the runtime for the same problem in Pyomo in section B.2. Pyomo uses one hundredth of the time to find a solution (which is more exact than the solution in any of the Dakota methods)! The computational cost of interfacing can be reduced (or maybe eliminated completely) by using Dakota's direct interface (which requires a custom installation) but there is probably not that much to gain when using Dakota on the types of problems it's built for: The effect of this overhead is greatly reduced when the simulation model is more complex than a simple polynomial function, as a larger proportion of the time is spent actually evaluating the model.

The problems tested in Pyomo or PySP can also be developed and analyzed in MATLAB. Solving LP, NLP and stochastic programs would however require additional packages to MATLAB like CASADi or similar. For experienced MATLAB users, MATLAB would work just fine for solving such mathematical optimization problems. However, based on research and results documented in this report, Pyomo is an excellent open source substitute to MATLAB.

The CPLEX solver that is used to solve the stochastic program in PySP has to be bought and downloaded from IBM's website. The license is free for students but not publicly available to other users. Even though Pyomo is an open-source software package, this restriction will however limit which solvers the user has access to and what kind of problems can be solved.

Bibliography

[1] Hart, William E., Carl D. Laird, Jean-Paul Watson, David L. Woodruff, Gabriel A. Hackebeil, Bethany L. Nicholson, and John D. Siirola.
Pyomo – Optimization Modeling in Python. Second Edition. Vol. 67. Springer, 2017.

[2] Pyomo website
`http://www.pyomo.org/about/`

[3] List of Optimization software
`https://en.wikipedia.org/wiki/List_of_optimization_software`

[4] Watson, Jean-Paul, David L. Woodruff, and William E. Hart. *PySP: modeling and solving stochastic programs in Python.* Mathematical Programming Computation 4, no. 2 (2012): 109-149.

[5] Adams, Brian M., et al. *DAKOTA, a multilevel parallel object-oriented framework for design optimization, parameter estimation, uncertainty quantification, and sensitivity analysis: version 6.7 user's manual* Sandia National Laboratories, Tech. Rep. SAND2014-4633 (2014 [updated 2017]).

[6] Ford, L. R.; Fulkerson, D. R. (1956). *Maximal flow through a network.* Canadian Journal of Mathematics. doi:10.4153/CJM-1956-045-5.

[7] Rosenbrock, H.H. (1960). *An automatic method for finding the greatest or least value of a function.* The Computer Journal. 3 (3): 175–184. doi:10.1093/comjnl/3.3.175. ISSN 0010-4620.

[8] Jeff Linderoth. *Stochastic Programming Modeling* `http://homepages.cae.wisc.edu/ linderot/classes/ie495/lecture3.pdf` University of Wisconsin-Madison. Published 20.01.2003

Appendix A

Download and Installation of Software

A.1 Pyomo

There are multiple ways to install Pyomo and its optimizing solvers. Homebrew and Anaconda are both package mangers aimed to install objects into `/usr/local/`. This downloading tutorial will be using Anaconda. Anaconda works for Linux, Mac OS/X and other Unix variants.Linux, Mac OS/X and other Unix variants typically have Python pre-installed[2]. To check what version you have, run the following command in your terminal:

```
1 $ python -V
```

Make sure to download the correct version of Anaconda for your Python version.

Install Pyomo with Anaconda

Download Anaconda from `https://www.anaconda.com/download/macos`. Run the following command to install the latest version of Pyomo:

```
1 $ conda install -c conda-forge pyomo
```

Pyomo also has conditional dependencies on a variety of third-party Python packages. PySP is part of this Pyomo extension package. These can also be installed with Anaconda:

```
1 $ conda install -c conda-forge pyomo.extras
```

Pyomo does not include any stand-alone optimization solvers. Consequently, most users will need to install third-party solvers to analyze optimization models built with Pyomo. The GNU Linear Programming Kit (GLPK) is a software package intended for solving large-scale linear

programming (LP), mixed integer programming (MIP), and other related problems. This software package can be downloaded with the following command in your terminal:

```
$ conda install -c bioconda glpk
```

IPOPT, short for "Interior Point OPTimizer, pronounced I-P-Opt", is a software library for large scale nonlinear optimization of continuous systems. This software library can be downloaded with the following command in your terminal:

```
$ conda install -c bioconda ipopt
```

A.2 Dakota

Dakota is available for Windows, Mac OS X and Linux. Downloading Dakota is a pretty straightforward process. A pre-built version of Dakota can be downloaded from Dakota's homepages https://dakota.sandia.gov/download.html (the pre-built Linux versions are built on RHEL 6 and 7, and might not work on other Linux distributions). Then, the files must be extracted to a fitting location, and the extracted directory must be renamed to "Dakota".

The platform's PATH must also be set to access the Dakota executables. This process is described in https://dakota.sandia.gov/content/set-environment-linux-mac-os-x (for OS X and Linux) and in https://dakota.sandia.gov/content/set-windows-environment (for Windows).

For advanced users, it's also possible to download and compile Dakota from source code.

Appendix B

Pyomo code

B.1 Command Line Arguments

- `-h or --help`

  Show help message and exit. The following command will show help message and other supporting commands to help with general problem solving.

  ```
  $ pyomo solve -h
  ```

  While this next command will provide support and messages for the glpk solver.

  ```
  $ pyomo solve --solver=glpk -h
  ```

  The next command will provide documentation about the different solver interfaces and solver managers available.

  ```
  $ pyomo help -s
  ```

- `--solver=`SOLVER TYPE

  The type of solver used to solve scenario sub-problems. Default is cplex. The following line will solve the instance specified in `someProblem.py` using a glpk solver. The glpk solver is most suitable for LP(linar programming and MIP (mixed integer programming).

  ```
  $ pyomo solve --solver=glpk someProblem.py
  ```

- Concrete Model

  `ConcreteProblem.py` contains both data and model instance. This command will solve the optimization problem in `ConcreteProblem.py`

  ```
  $ pyomo solve --solver=somesolver ConcreteProblem.py
  ```

- Abstract Model

  `AbstractProblem.py` contains the model instance while `AbstractProblem.dat` contains the supporting data. This command will solve the optimization problem that is defined through `AbstractProblem.py` and `AbstractProblem.dat`

  ```
  $ pyomo solve --solver=somesolver AbstractProblem.py AbstractProblem.dat
  ```

B.2 Rosebrock Function

Command line

The following command is used in the terminal to compile and execute the optimization problem.

```
$ pyomo solve --solver=ipopt rosenbrock.py
```

Rosenbrock.py

```
# 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(model):
```

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

Rosenbrock result

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Lower bound: -inf
  Upper bound: inf
  Number of objectives: 1
  Number of constraints: 0
  Number of variables: 2
  Sense: unknown
# ----------------------------------------------------------
#   Solver Information
# ----------------------------------------------------------
Solver:
- Status: ok
  Message: Ipopt 3.11.1\x3a Optimal Solution Found
  Termination condition: optimal
  Id: 0
  Error rc: 0
  Time: 0.021049976348876953
# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
```

```
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: None
  Status: optimal
  Message: Ipopt 3.11.1\x3a Optimal Solution Found
  Objective:
    obj:
      Value: 7.013645951336496e-25
  Variable:
    y:
      Value: 1.0000000000016314
    x:
      Value: 1.0000000000008233
  Constraint: No values
```

B.3 Maximum flow problem

This is code for the maximum flow problem example. In this case *Zoo* is the source and *Home* is the source.

Command line

The following command is used in the terminal to compile and execute the optimization problem.

```
$ pyomo solve --solver=glpk maxflow.py maxflow.dat
```

maxflow.dat

```
set N := Zoo A B C D E Home;
set A := (Zoo,A) (Zoo,B) (A,C) (A,D) (B,A) (B,C) (C,D) (C,E) (D,E) (D,Home) (E,Home);

param s := Zoo;
param t := Home;
param: c :=
Zoo A 11
Zoo B 8
A C 5
A D 8
B A 4
B C 3
C D 2
C E 4
D E 5
D Home 8
E Home 6;
```

maxflow.py

```
from pyomo.environ import *

model = AbstractModel()

# Nodes in the network
model.N = Set()

# Network arcs
model.A = Set(within=model.N*model.N)

# Source node
model.s = Param(within=model.N)

# Sink node
model.t = Param(within=model.N)

# Flow capacity limits
model.c = Param(model.A)

# The flow over each arc
model.f = Var(model.A, within=NonNegativeReals)

# Maximize the flow into the sink nodes
def total_rule(model):
    return sum(model.f[i,j] for (i, j) in model.A if j == value(model.t))
model.total = Objective(rule=total_rule, sense=maximize)

# Enforce an upper limit on the flow across each arc
def limit_rule(model, i, j):
```

```
    return model.f[i,j] <= model.c[i, j]
model.limit = Constraint(model.A, rule=limit_rule)

# Enforce flow through each node
def flow_rule(model, k):
    if k == value(model.s) or k == value(model.t):
        return Constraint.Skip
    inFlow  = sum(model.f[i,j] for (i,j) in model.A if j == k)
    outFlow = sum(model.f[i,j] for (i,j) in model.A if i == k)
    return inFlow == outFlow
model.flow = Constraint(model.N, rule=flow_rule)
```

Maxflow result

```
# ==========================================================
# = Solver Results                                         =
# ==========================================================
# ----------------------------------------------------------
#   Problem Information
# ----------------------------------------------------------
Problem:
- Name: unknown
  Lower bound: 14.0
  Upper bound: 14.0
  Number of objectives: 1
  Number of constraints: 17
  Number of variables: 12
  Number of nonzeros: 30
  Sense: maximize
# ----------------------------------------------------------
#   Solver Information
```

```
# ----------------------------------------------------------
Solver:
- Status: ok
  Termination condition: optimal
  Statistics:
    Branch and bound:
      Number of bounded subproblems: 0
      Number of created subproblems: 0
  Error rc: 0
  Time: 0.023765087127685547
# ----------------------------------------------------------
#   Solution Information
# ----------------------------------------------------------
Solution:
- number of solutions: 1
  number of solutions displayed: 1
- Gap: 0.0
  Status: feasible
  Message: None
  Objective:
    total:
      Value: 14
  Variable:
    f[D,E]:
      Value: 2
    f[D,Home]:
      Value: 8
    f[B,C]:
      Value: 3
    f[Zoo,A]:
```

```
    Value: 11
  f[E,Home]:
    Value: 6
  f[C,E]:
    Value: 4
  f[C,D]:
    Value: 2
  f[A,C]:
    Value: 3
  f[A,D]:
    Value: 8
  f[Zoo,B]:
    Value: 3
Constraint: No values
```

Appendix C

PySP code

C.1 Command Line Arguments

These are some useful command line arguments for PySP:

- `--model-directory`=MODEL DIRECTORY

  The directory of the reference model file `ReferenceModel.py` and `ReferenceModel.dat`

- `--instance-directory`=SCENARIO DIRECTORY

  The directory where the scenario tree is specified. The directory of `ScenarioStructure.py` and `ScenarioStructure.dat` as well as data for the particular scenarios.

- `--solver`=SOLVER TYPE

  The type of solver used to solve scenario sub-problems. Default is cplex.

To compile and run the farmer problem, type the following command with cplex solver in the terminal:

```
$ runef --model-directory=MODELDIRECTORY --instance-directory=SCENARIODATA
```

C.2 Farmer problem

ReferenceModel.py

```
1  ___________________________________________________________________________
2  #
3  #  Pyomo: Python Optimization Modeling Objects
4  #  Copyright 2017 National Technology and Engineering Solutions of Sandia, LLC
5  #  Under the terms of Contract DE-NA0003525 with National Technology and
6  #  Engineering Solutions of Sandia, LLC, the U.S. Government retains certain
7  #  rights in this software.
8  #  This software is distributed under the 3-clause BSD License.
9  #  ___________________________________________________________________________
10
11 # Farmer: rent out version has a scalar root node var
12 # note: this will minimize
13 #
14 # Imports
15 #
16
17 from pyomo.core import *
18
19 #
20 # Model
21 #
22
23 model = AbstractModel()
24
25 #
26 # Parameters
27 #
28
29 model.CROPS = Set()
30
31 model.TOTAL_ACREAGE = Param(within=PositiveReals)
32
33 model.PriceQuota = Param(model.CROPS, within=PositiveReals)
34
35 model.SubQuotaSellingPrice = Param(model.CROPS, within=PositiveReals)
36
```

```
37 def super_quota_selling_price_validate (model, value, i):
38     return model.SubQuotaSellingPrice[i] >= model.SuperQuotaSellingPrice[i]
39
40 model.SuperQuotaSellingPrice = Param(model.CROPS, validate=
       super_quota_selling_price_validate)
41
42 model.CattleFeedRequirement = Param(model.CROPS, within=NonNegativeReals)
43
44 model.PurchasePrice = Param(model.CROPS, within=PositiveReals)
45
46 model.PlantingCostPerAcre = Param(model.CROPS, within=PositiveReals)
47
48 model.Yield = Param(model.CROPS, within=NonNegativeReals)
49
50 #
51 # Variables
52 #
53
54 model.DevotedAcreage = Var(model.CROPS, bounds=(0.0, model.TOTAL_ACREAGE))
55
56 model.QuantitySubQuotaSold = Var(model.CROPS, bounds=(0.0, None))
57 model.QuantitySuperQuotaSold = Var(model.CROPS, bounds=(0.0, None))
58
59 model.QuantityPurchased = Var(model.CROPS, bounds=(0.0, None))
60
61 #
62 # Constraints
63 #
64
65 def ConstrainTotalAcreage_rule(model):
66     return summation(model.DevotedAcreage) <= model.TOTAL_ACREAGE
67
68 model.ConstrainTotalAcreage = Constraint(rule=ConstrainTotalAcreage_rule)
69
70 def EnforceCattleFeedRequirement_rule(model, i):
71     return model.CattleFeedRequirement[i] <= (model.Yield[i] * model.DevotedAcreage[i]) +
```

```
    model.QuantityPurchased[i] - model.QuantitySubQuotaSold[i] - model.
    QuantitySuperQuotaSold[i]
72
73 model.EnforceCattleFeedRequirement = Constraint(model.CROPS, rule=
    EnforceCattleFeedRequirement_rule)
74
75 def LimitAmountSold_rule(model, i):
76     return model.QuantitySubQuotaSold[i] + model.QuantitySuperQuotaSold[i] - (model.Yield
    [i] * model.DevotedAcreage[i]) <= 0.0
77
78 model.LimitAmountSold = Constraint(model.CROPS, rule=LimitAmountSold_rule)
79
80 def EnforceQuotas_rule(model, i):
81     return (0.0, model.QuantitySubQuotaSold[i], model.PriceQuota[i])
82
83 model.EnforceQuotas = Constraint(model.CROPS, rule=EnforceQuotas_rule)
84
85 #
86 # Stage-specific cost computations
87 #
88
89 def ComputeFirstStageCost_rule(model):
90     return summation(model.PlantingCostPerAcre, model.DevotedAcreage)
91
92 model.FirstStageCost = Expression(rule=ComputeFirstStageCost_rule)
93
94 def ComputeSecondStageCost_rule(model):
95     expr = summation(model.PurchasePrice, model.QuantityPurchased)
96     expr -= summation(model.SubQuotaSellingPrice, model.QuantitySubQuotaSold)
97     expr -= summation(model.SuperQuotaSellingPrice, model.QuantitySuperQuotaSold)
98     return expr
99
100 model.SecondStageCost = Expression(rule=ComputeSecondStageCost_rule)
101
102 #
103 # PySP Auto-generated Objective
```

```
104 #
105 # minimize: sum of StageCosts
106 #
107 # An active scenario objective equivalent to that generated by PySP is
108 # included here for informational purposes.
109 def total_cost_rule(model):
110     return model.FirstStageCost + model.SecondStageCost
111 model.Total_Cost_Objective = Objective(rule=total_cost_rule, sense=minimize)
```

ReferenceModel.dat

```
1 set CROPS := WHEAT CORN SUGAR_BEETS ;
2
3 param TOTAL_ACREAGE := 500 ;
4
5 # no quotas on wheat or corn
6 param PriceQuota :=
7             WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;
8
9 param SubQuotaSellingPrice :=
10            WHEAT 170 CORN 150 SUGAR_BEETS 36 ;
11
12 param SuperQuotaSellingPrice :=
13            WHEAT 0 CORN 0 SUGAR_BEETS 10 ;
14
15 param CattleFeedRequirement :=
16            WHEAT 200 CORN 240 SUGAR_BEETS 0 ;
17
18 # can't purchase beets (no need, as cattle don't eat them)
19 param PurchasePrice :=
20            WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;
21
22 param PlantingCostPerAcre :=
23            WHEAT 150 CORN 230 SUGAR_BEETS 260 ;
24
25 param Yield := WHEAT 3.0 CORN 3.6 SUGAR_BEETS 24 ;
```

ScenarioStructure.py

```
from coopr.pyomo import *
model = AbstractModel()
model.Stages = Set(ordered=True)
model.Nodes = Set()
model.NodeStage = Param(model.Nodes, \
within=model.Stages)
model.Children = Set(model.Nodes, \
within=model.Nodes, \
ordered=True)
model.ConditionalProbability = Param(model.Nodes)
model.Scenarios = Set(ordered=True)
model.ScenarioLeafNode = Param(model.Scenarios, \
within=model.Nodes)
model.StageVariables = Set(model.Stages)
model.StageCostVariable = Param(model.Stages)
model.ScenarioBasedData = Param(within=Boolean, default=True)
```

ScenarioStructure.dat

```
# IMPORTANT - THE STAGES ARE ASSUMED TO BE IN TIME-ORDER.

set Stages := FirstStage SecondStage ;

set Nodes := RootNode
             BelowAverageNode
             AverageNode
             AboveAverageNode ;

param NodeStage := RootNode         FirstStage
                   BelowAverageNode SecondStage
                   AverageNode      SecondStage
                   AboveAverageNode SecondStage ;

set Children[RootNode] := BelowAverageNode
                          AverageNode
```

```
17                                      AboveAverageNode  ;
18
19 param ConditionalProbability := RootNode            1.0
20                                  BelowAverageNode   0.33333333
21                                  AverageNode        0.33333334
22                                  AboveAverageNode   0.33333333  ;
23
24 set Scenarios := BelowAverageScenario
25                  AverageScenario
26                  AboveAverageScenario  ;
27
28 param ScenarioLeafNode :=
29                      BelowAverageScenario  BelowAverageNode
30                      AverageScenario       AverageNode
31                      AboveAverageScenario  AboveAverageNode  ;
32
33 set StageVariables[FirstStage] :=  DevotedAcreage[*];
34
35 set StageVariables[SecondStage] := QuantitySubQuotaSold[*]
36                                    QuantitySuperQuotaSold[*]
37                                    QuantityPurchased[*];
38
39 param StageCost := FirstStage   FirstStageCost
40                    SecondStage  SecondStageCost  ;
```

AboveAverageScenario.dat

```
1
2 # above mean scenario
3
4 set CROPS := WHEAT CORN SUGAR_BEETS  ;
5
6 param TOTAL_ACREAGE := 500  ;
7
8 # no quotas on wheat or corn
9 param PriceQuota := WHEAT 100000 CORN 100000 SUGAR_BEETS 6000  ;
10
```

```
param SubQuotaSellingPrice := WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice := WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement := WHEAT 200 CORN 240 SUGAR_BEETS 0 ;

# can't purchase beets (no real need, as cattle don't eat them)
param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 3.0 CORN 3.6 SUGAR_BEETS 24 ;
```

BelowAverageScenario.dat

```
# below-mean scenario

set CROPS := WHEAT CORN SUGAR_BEETS ;

param TOTAL_ACREAGE := 500 ;

# no quotas on wheat or corn
param PriceQuota := WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;

param SubQuotaSellingPrice := WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice := WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement := WHEAT 200 CORN 240 SUGAR_BEETS 0 ;

# can't purchase beets (no real need, as cattle don't eat them)
param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 2.0 CORN 2.4 SUGAR_BEETS 16 ;
```

AverageScenario.dat

```
# "mean" scenario

set CROPS := WHEAT CORN SUGAR_BEETS ;

param TOTAL_ACREAGE := 500 ;

# no quotas on wheat or corn
param PriceQuota := WHEAT 100000 CORN 100000 SUGAR_BEETS 6000 ;

param SubQuotaSellingPrice := WHEAT 170 CORN 150 SUGAR_BEETS 36 ;

param SuperQuotaSellingPrice := WHEAT 0 CORN 0 SUGAR_BEETS 10 ;

param CattleFeedRequirement := WHEAT 200 CORN 240 SUGAR_BEETS 0 ;

# can't purchase beets (no real need, as cattle don't eat them)
param PurchasePrice := WHEAT 238 CORN 210 SUGAR_BEETS 100000 ;

param PlantingCostPerAcre := WHEAT 150 CORN 230 SUGAR_BEETS 260 ;

param Yield := WHEAT 2.5 CORN 3 SUGAR_BEETS 20 ;
```

Appendix D

Dakota code

D.1 Dakota

To run Dakota, you need open the command line (remember to set the path!), navigate to the directory in which your input file is located, then write:

```
dakota [name of input file].in
```

This will just run the actions in the in-file, and show the output in the console. To save the output to a textfile, you can write

```
dakota -i [name of input file].in -o [name of output file].out
```

If you include the `tabular_data` option in your input file (as shown below in section D.2), the raw data will be saved in a .dat file, which is generally more useful than saving the console output like above.

D.2 Rosenbrock Function in Python

```
1 # Demonstrate Dakota script interface to Python analysis (Rosenbrock)
2
3 environment,
4   tabular_data
5     tabular_data_file = 'rosen_opt.dat'
6
7 method,
```

```
8    optpp_newton
9      max_iterations = 50
10     convergence_tolerance = 1e-4
11
12 model,
13   single
14
15 variables,
16   continuous_design = 2
17     cdv_initial_point   -1.2   1.0
18     cdv_lower_bounds    -2.0  -2.0
19     cdv_upper_bounds     2.0   2.0
20     cdv_descriptor       'x1'  'x2'
21
22 interface,
23   fork
24    analysis_drivers = 'python.exe rosenbrock_bb_di.py'
25    parameters_file = 'params.in'
26    results_file = 'results.out'
27
28 responses,
29   num_objective_functions = 1
30   analytic_gradients
31   analytic_hessians
```

The wrapper function `rosenbrock_bb_di.py` is specified in the `analysis_drivers` field.

```
1 import dakota.interfacing as di
2
3 # ----------------------------
4 # Parse Dakota parameters file
5 # ----------------------------
6
7 params, results = di.read_parameters_file()
8
9 # -------------------------------
10 # Convert and send to application
```

```
11 # ---------------------------------------------
12
13
14 # set up the data structures the rosenbrock analysis code expects
15 # for this simple example, put all the variables into a single hardwired array
16 # The asv has to be mapped back into an integer
17 continuous_vars = [ params['x1'], params['x2'] ]
18 active_set_vector = 0
19 if results["obj_fn"].asv.function:
20     active_set_vector += 1
21 if results["obj_fn"].asv.gradient:
22     active_set_vector += 2
23 if results["obj_fn"].asv.hessian:
24     active_set_vector += 4
25
26 # set a dictionary for passing to rosenbrock via Python kwargs
27 rosen_params = {}
28 rosen_params['cv'] = continuous_vars
29 rosen_params['asv'] = [active_set_vector]
30 rosen_params['functions'] = 1
31
32 # execute the rosenbrock analysis as a separate Python module
33 print "Running rosenbrock..."
34 from rosenbrock import rosenbrock_list
35 rosen_results = rosenbrock_list(**rosen_params)
36 print "Rosenbrock complete."
37
38 # ----------------------------------------
39 # Return the results to Dakota
40 # ----------------------------------------
41
42 # Insert functions from rosen into results
43 # Results iterator provides an index, response name, and response
44 for i, n, r in results:
45     if r.asv.function:
46         r.function = rosen_results['fns'][i]
```

```
47      if r.asv.gradient:
48          r.gradient = rosen_results['fnGrads'][i]
49      if r.asv.hessian:
50          r.hessian = rosen_results['fnHessians'][i]
51
52  results.write()
```

The wrapper function calls the actual simulation model. In this case it is simply the Rosenbrock function, which is a test function for optimization algorithms. It could be swapped out for a complex model with the same number of inputs and outputs without needing to change much (if anything) in the input- and wrapper files.

```
1   def rosenbrock_list(**kwargs):
2
3       num_fns = kwargs['functions']
4
5       x = kwargs['cv']
6       ASV = kwargs['asv']
7
8       f0 = x[1]-x[0]*x[0]
9       f1 = 1-x[0]
10
11      retval = dict([])
12
13      if (ASV[0] & 1): # **** f:
14          f = [100*f0*f0+f1*f1]
15          retval['fns'] = f
16
17      if (ASV[0] & 2): # **** df/dx:
18          g = [ [-400*f0*x[0] - 2*f1, 200*f0] ]
19          retval['fnGrads'] = g
20
21      if (ASV[0] & 4): # **** d^2f/dx^2:
22          fx = x[1]-3*x[0]*x[0]
23
24          h = [
25                [ [-400*fx + 2, -400*x[0]],
```

```
26                [-400*x[0],     200      ] ]
27            ]
28        retval['fnHessians'] = h
29
30    return(retval)
```

D.3 Rosenbrock Function in MATLAB

Another example of an input file, this time interfacing with MATLAB. Note that this input file is without the `environment` block as well as the `model` block. The lack of a `model` block makes no difference, as the default value is `single`, like specified above. The lack of the `environment` block means no .dat file will be created. A different local gradient based solver than before is used (`conmin_frcg`) and the gradients are estimated numerically. In reality, the `interface` block from this input file could be inserted into the one above (assuming that the MATLAB model would calculate the gradients and hessians, which this example does not), and it would work just as well.

```
1  method,
2    conmin_frcg
3
4  variables,
5    continuous_design = 2
6    cdv_initial_point    -1.0     1.0
7    cdv_lower_bounds     -2.0    -2.0
8    cdv_upper_bounds      2.0     2.0
9    cdv_descriptor        'x1'    'x2'
10
11 interface,
12   fork,
13     parameters_file = 'params.in'
14     results_file    = 'results.out'
15     analysis_driver = 'cscript matlab_rosen_bb_simulator.vbs'
16
17 responses,
18   num_objective_functions = 1
```

```
19   numerical_gradients
20     fd_gradient_step_size = .000001
21   no_hessians
```

Note that the input file calls a VBScript, which in turn calls the MATLAB wrapper script:

```
1 Dim mlobj
2
3 Set mlobj = CreateObject("Matlab.Application")
4
5 ' Direct the matlab server to change to the current working directory, where the
6 ' parameters file will be written by Dakota, and where Dakota expects to find
7 ' the responses file after completion of this script
8 sCurPath = CreateObject("Scripting.FileSystemObject").GetAbsolutePathName(".")
9 cd_command = "cd('" & sCurPath & "')"
10 mlobj.Execute(cd_command)
11
12 'Get the command line arguments passed by DAKOTA, so we work on the correct files
13 paramsfile  = WScript.Arguments.Item(0)
14 resultsfile = WScript.Arguments.Item(1)
15
16 Dim matlab_command
17 matlab_command = "matlab_rosen_wrapper('" & paramsfile & "', '" & resultsfile & "')"
18
19 Dim result
20 result = mlobj.Execute(matlab_command)
21 WScript.Echo "Matlab returned: " & result
22
23 'TODO: capture and act on any error from Matlab
```

The MATLAB wrapper is:

```
1 function matlab_rosen_wrapper(params, results)
2
3 %--------------------------------------------------------------
4 % Set any fixed/default values needed for your analysis .m code
5 %--------------------------------------------------------------
6
7 alpha = 100;
```

```
8
9
10 %------------------------------------------------------------------------
11 % READ params.in (or params.in.num) from DAKOTA and set Matlab variables
12 %
13 % read params.in (no aprepro) -- just one param per line
14 % continuous design, then U.C. vars
15 % --> 1st cell of C has values, 2nd cell labels
16 % --> variables are in rows 2-->?
17 %------------------------------------------------------------------------
18
19 fid = fopen(params,'r');
20 C = textscan(fid,'%n%s');
21 fclose(fid);
22
23 num_vars = C{1}(1);
24
25
26 % set design variables -- could use vector notation
27 % rosenbrock x1, x2
28
29 x(1) = C{1}(2);
30 x(2) = C{1}(3);
31
32
33 %------------------------------------------------------------------------
34 % CALL your analysis code to get the function value
35 %------------------------------------------------------------------------
36
37 [f] = rosenbrock(x,alpha);
38
39
40 %------------------------------------------------------------------------
41 % WRITE results.out
42 %------------------------------------------------------------------------
43 fid = fopen(results,'w');
```

```
44 fprintf(fid,'%20.10e        f\n', f);
45 %fprintf(fid,'%20.10e        params\n', C{1}(2:5));
46
47 fclose(fid);
48
49 % alternately
50 %save(results,'vContact','-ascii');
```

The actual objective function is (like before) very simple, but could just as well be a complicated simulation model.

```
1 % return function values for the rosenbrock problem
2 %    f( x1, x2 ) = alpha*( x2 - x1^2 )^2 + ( 1 - x1 )^2
3
4 function f = rosenbrock(x,alpha);
5
6   f = alpha*( x(2) - x(1)^2 )^2 + ( 1 - x(1) )^2;
7
8 end
```