Chapter 2
Mathematical Modeling and Optimization

Abstract This chapter provides a primer on optimization and mathematical modeling. It does not provide a complete description of these topics. Instead, this chapter provides enough background information to support reading the rest of the book. For more discussion of optimization modeling techniques see, for example, Williams [86]. Implementations of simple examples of models are shown to provide the reader with a quick start to using Pyomo.

2.1 Mathematical Modeling

2.1.1 Overview

Modeling is a fundamental process in many aspects of scientific research, engineering, and business. Modeling involves the formulation of a simplified representation of a system or real-world object. These simplifications allow structured representation of knowledge about the original system that facilitates the analysis of the resulting model. Schichl [78] notes that models are used to

- **Explain phenomena** that arise in a system;
- **Make predictions** about future states of a system;
- **Assess key factors** that influence phenomena in a system;
- **Identify extreme states** in a system that might represent worst-case scenarios or minimal cost plans; and
- **Analyze trade-offs** to support human decision makers.

Additionally, the structured aspect of a model's representation facilitates communication of the knowledge associated with a model. For example, a key aspect of a model is its level of detail, which reflects the system knowledge that is needed to employ the model in an application.

Mathematics has always played a fundamental role in representing and formulating our knowledge. Mathematical modeling has become increasingly formal as new

W.E. Hart et al., *Pyomo — Optimization Modeling in Python*, Springer Optimization and Its Applications 67, DOI 10.1007/978-3-319-58821-6_2

frameworks have emerged to express complex systems. The following mathematical concepts are central to modern modeling activities:

- **Variables**: These represent *unknown* or changing parts of a model (e.g., which decisions to take, or the characteristic of a system outcome).
- **Parameters**: These are symbolic representations for real-world data, which might vary for different problem instances or scenarios.
- **Relations**: These are *equations*, *inequalities*, or other mathematical relationships that define how different parts of a model are related to each other.

Optimization models are mathematical models that include functions that represent goals or objectives for the system being modeled. Optimization models can be analyzed to explore system trade-offs in order to find solutions that optimize system objectives. Consequently, these models can be used for a wide range of scientific, business, and engineering applications.

2.1.2 A Modeling Example

A *Model*, in the sense that we will use the word, represents items by abstracting away some features. Everyone is familiar with physical models, such as model railroads or model cars. Our interest is in mathematical models that use symbols to represent aspects of a system or real-world object.

For example, a person might want to determine the best number of scoops of ice cream to buy. We could use the symbol x to represent the number of scoops. We might use c to represent the cost per scoop. So then we could model the total cost as c times x, which we usually write as cx.

We might need a more sophisticated model of total cost if there are volume discounts or surcharges for buying fractional scoops. Also, this model is probably not valid for negative values of x. It is seldom possible to sell back ice cream for the same price paid for it.

It is more complicated to provide a mathematical model of the happiness associated with scoops of ice cream on an ice cream cone. One approach is to use a scaled measure of happiness. We will do that using the basic unit of the happiness associated with one scoop of ice cream, which we call h. A simple model, then, would be to say that the total happiness from x scoops of ice cream is h times x, which we write as hx. For some people, that might be a pretty good approximation for values of x between one-half and three, but there is almost no one who is 100 times as happy to have 100 scoops of ice cream on their ice cream cone as they are to have one scoop. For some people, the model of happiness for values of x between zero and ten might be something like

$$h \cdot \left(x - (x/5)^2\right).$$

Note that this model becomes negative when there are more than 25 scoops on the cone, which might not be a good model for everyone.

It is common to want to model more than one thing at a time. For example, you might be able to have scoops of ice cream and peanuts. Since there are multiple things that can be purchased, we can represent the quantity purchased using a vector x (i.e. the symbol x now represents a list). We refer to *elements* of the list using the notation x_i where the symbol i indexes the vector. For example, if we agree that the first element is the number of scoops of ice cream, then this number could be referenced using x_1. For higher dimensions we use a *tuple*, such as i, j or (i, j) as the index.

Let's change c to be a vector of costs with the same indices as x (i.e., c_1 is the cost per scoop of ice cream and c_2 is the cost per cup of peanuts). So now, we write the total cost of ice cream and peanuts as

$$c_1x_1 + c_2x_2 = \sum_{i=1}^{2} c_ix_i.$$

Once again, this cost model is probably not valid for all possible values of all elements of x, but it might be good enough for some purposes.

Often, it is useful to refer to indices as being members of a set. For the example just given, we could use the set $\{1,2\}$ to write the total cost as

$$\sum_{i\in\{1,2\}} c_ix_i.$$

but it would be more common to use a more abstract expression like

$$\sum_{i\in\mathscr{A}} c_ix_i$$

where the set $\mathscr{A}$ is understood to be the index set for c and x (and for our example the set $\mathscr{A}$ would be $\{1,2\}$.)

In addition to summing over an index set, we might want to have conditions that hold for all members of an index set. This is done simply by using a comma. For example, if we want to require that none of the values of x can be negative, we would write

$$x_i \geq 0,\ i \in \mathscr{A}$$

and we read this line out loud as "x subscript i is greater than or equal to zero for all i in A."

There is no law of mathematics or even mathematical modeling that requires the use of single letter symbols such as x and c or i. It would be perfectly okay for the set $\mathscr{A}$ to be composed of a picture of an ice cream cone and a picture of a cup of peanuts, but that is hard to work with in some settings. The set could also be $\{Scoops, Cups\}$, but that is not commonly done in books because it takes up too much space and causes lines to overflow. Also, x could be replaced by something like $Quantity$. Long names are, importantly, supported by modeling languages such as Pyomo, and it is generally a good idea to use meaningful names when writing Pyomo models. Spaces or dashes embedded in names often cause troubles and confusion, so underscores

are often used in long names instead.

2.2 Optimization

The symbol x is often used as a *variable* in optimization modeling. It is sometimes called a *decision variable* because we build optimization models to help make decisions. This can sometimes cause a little confusion for people who are familiar with modeling as practiced by statisticians. They often use the symbol x to refer to data. Thus statisticians give values of x to the computer to have it compute statistics, while optimization modelers give other data to the computer and ask the computer to compute good values of x. Of course, symbols other than x can be used; however, in text books and introductions x is often chosen.

Values such as cost (we used the symbol c) are referred to as *data* or *parameters*. An optimization model can be described with undefined parameter values, but a specific instance that is optimized must have specific data values, which we sometimes call *instance data*.

A model must have an objective to perform optimization, which is expressed as an *objective function*. Optimal values of the decision variables result in *the* best possible value of the objective function. It is important to note that we did not say "the optimal values" because it is often the case that more than one set of variable values result in the best possible value of the objective function. It is common to write this function in a very abstract way, such as $f(x)$. Whether the best is the smallest or the largest possible value is determined by the *sense* of the optimization: *minimize* or *maximize*.

For example, suppose that x is not a vector, but rather a *scalar* that denotes the number of scoops of ice cream to buy. If we use the model of happiness given before, then

$$f(x) \equiv h \cdot \left(x - (x/5)^2\right),$$

where h is given as data. (It turns out not to matter what value of h is given for the purpose of finding the x that maximizes happiness in this particular example.) The optimization problem, as we have modeled it, is given as

$$\max\ h \cdot \left(x - (x/5)^2\right),$$

but very careful authors would write

$$\max_x\ h \cdot \left(x - (x/5)^2\right)$$

to make it clear that x is the decision variable. In this case, there is only one best value of x, which can be found using numerical optimization. The best value of x turns out to be fractional, which means that it is not an integer number of scoops. This model might not be considered useful for a typical ice cream shop, where the number of scoops must be a non-negative integer. To specify this requirement, we

add a *constraint* to the optimization model:

$$\begin{aligned} \max_x \; & h \cdot \left(x - (x/5)^2\right) \\ \text{s.t.} \; & \\ & x \in \text{non-negative integers} \end{aligned}$$

where "s.t." is an abbreviation for either "subject to" or "such that." Suppose that the model is not being used in an ice cream shop, but rather at home, where the ice cream is being served by the model user's parent. If the parent is willing to make partial scoops but not willing to go above two scoops, then the constraint

$$x \in \text{non-negative integers}$$

would be replaced with

$$0 \leq x \leq 2.$$

This is not a perfect model because really, not all fractional values of x would be reasonable.

To illustrate the model aspects discussed so far, let us return to multiple products described by an index set $\mathscr{A}$, so x is a vector. Let us make use of the following model of happiness for a product index i:

$$h_i \cdot \left(x_i - (x_i/d_i)^2\right),$$

where h and d are both data vectors with the same index set as x. Further, let c be a vector of costs and u be a vector of the most of any product that can be purchased. Let us assume that all products can be purchased in fractional quantities for the moment. Finally, suppose there is a total budget given by b. The optimization problem would be written as:

$$\begin{aligned} \max_x \; & \sum_{i \in \mathscr{A}} h_i \cdot \left(x_i - (x_i/d_i)^2\right) \quad (H) \\ \text{s.t.} \; & \sum_{i \in \mathscr{A}} c_i x_i \leq b \\ & 0 \leq x_i \leq u_i, \;\; i \in \mathscr{A} \end{aligned}$$

Some modelers would express the last constraint separately:

$$\begin{aligned} \max_x \; & \sum_{i \in \mathscr{A}} h_i \cdot \left(x_i - (x_i/d_i)^2\right) \quad (H) \\ \text{s.t.} \; & \sum_{i \in \mathscr{A}} c_i x_i \leq b \\ & x_i \leq u_i, i \in \mathscr{A} \\ & x_i \geq 0, i \in \mathscr{A} \end{aligned}$$

It is common to put a short, abbreviated name of the model in parentheses on the same line as the objective. The name (P) is very common, but we used (H) as a mnemonic for "happiness." The name (H) allows us to refer to this model later in the chapter, where we show how to implement it in Pyomo and solve it.

2.3 Linear and Nonlinear Optimization Models

2.3.1 Definition

An expression in an optimization model is said to be linear if it is composed only of sums of decision variables and/or decision variables multiplied by data. Thus, a linear expression is an expression that is a non-constant, linear function of the decision variables. Assume that x is a variable vector, c is a vector of data and that both are indexed by $\mathscr{A}$. Further assume that 2 and 3 are members of $\mathscr{A}$. The following are linear expressions:

$$\begin{array}{l}\sum_{i\in\mathscr{A}} c_i x_i \\ \sum_{i\in\mathscr{A}} x_i \\ x_2 \\ c_3 x_2 + c_2 x_3 \\ c_3 x_2 + c_2 x_3 + 4\end{array}$$

On the other hand, the following expressions are not linear: x_i^2, $x_2 x_3$ and cosine(x_2).

Linear expressions often result in problems that can be solved with much less computational effort than similar models with nonlinear expressions. Consequently, many modelers make an effort to use linear expressions as much as possible, and some modelers strive to use only linear expressions. Additionally, many models develop linear approximations to nonlinear models in hopes of finding "good enough" solutions to the original nonlinear model. In the context of nonlinear models, those with nonlinear objective functions are typically easier to optimize than models with nonlinear constraint expressions.

2.3.2 A Linear Approximation

We consider a simple way to linearize (H) to illustrate a linear optimization model. We describe a version of (H) that does not have a squared objective, but that is somewhat similar to (H).

For the moment, suppose that x is not an indexed list. We call x a *scalar* value. The nonlinearities lie in the expression

$$h \cdot \left(x - (x/5)^2\right),$$

and in (H) we generalize this to be of the form:

$$h \cdot \left(x - (x/d)^2\right).$$

To simplify matters, let us require x to be non-negative and less than or equal to u. We can form a linear expression that has the function value correctly computed at the endpoints, namely zero and u. The linear function will connect these points with a line. This expression is zero when x is zero, and it is $h \cdot \left(u - (u/d)^2\right)$ when x=u. The slope of the line between these two points is

$$\frac{h \cdot \left(u - (u/d)^2\right) - 0}{u - 0} = h \cdot \left(1 - u/d^2\right).$$

So a simple approximation for $h \cdot \left(x - (x/d)^2\right)$ on the interval $[0, u]$ is

$$h \cdot \left(1 - u/d^2\right) x.$$

As can be seen in Figure 2.1, the linear approximation is quite good for the ice cream example when $h = 1$, $d = 5$ and $u = 5$. In contrast, this simple method would work very poorly if $u = 25$ for the same value of d.

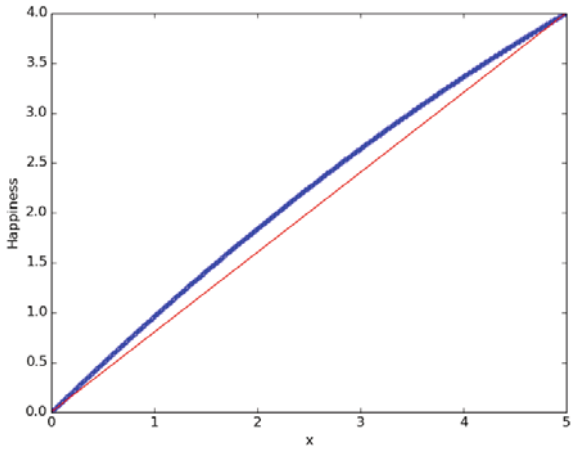

Fig. 2.1: Plotting the quadratic happiness function (the thicker line) and the linear approximation (the thin line). Both lines are drawn for $h = 1$, $d = 5$, and $u = 5$.

For illustrative purposes, to construct an expedient linear approximation to (H) we replace the objective function with

$$\max_x \sum_{i \in \mathscr{A}} h_i \cdot \left(1 - u/d_i^2\right) x_i \tag{2.1}$$

We say that this expression is linear because the decision variables are only multiplied by data, and summed. It is true that the parameter d is squared, but this is not a decision variable. The numerical value of the entire expression

$$h_i \cdot \left(1 - u_i / d_i^2\right)$$

is computed by Pyomo before the problem instance is passed to a solver, and the task of the solver is to find values that are optimal for the decision variables.

2.4 Modeling with Pyomo

We now consider different strategies for formulating and optimizing algebraic optimization models using Pyomo. Although a detailed explanation of Pyomo models is deferred to Chapter 3, the following examples illustrate the use of Pyomo for model (H).

2.4.1 An Abstract Formulation

An *abstract* mathematical formulation relies on unspecified parameter values. Since model (H) is an abstract model, a natural way of expressing this model in Pyomo is with Pyomo's `AbstractModel` class. A Pyomo `AbstractModel` defers initialization of model components until a *model instance* is created using set and parameter data. Thus, this modeling approach closely reflects the character of an abstract model.

Consider the following abstract Pyomo model for model (H):

```
# AbstractH.py - Implement model (H)
from pyomo.environ import *

model = AbstractModel(name="(H)")

model.A = Set()

model.h = Param(model.A)
model.d = Param(model.A)
model.c = Param(model.A)
model.b = Param()
model.u = Param(model.A)

def xbounds_rule(model, i):
   return (0, model.u[i])
model.x = Var(model.A, bounds=xbounds_rule)

def obj_rule(model):
   return sum(model.h[i] * \
```

```
            (model.x[i] - (model.x[i]/model.d[i])**2) \
            for i in model.A)
model.z = Objective(rule=obj_rule, sense=maximize)

def budget_rule(model):
   return sum(model.c[i]*model.x[i] for i in model.A) <= \
       model.b
model.budgetconstr = Constraint(rule=budget_rule)
```

Given particular data for the parameters in this model, then one might be interested in finding an optimal assignment of values to `model.x`. There are numerous ways to provide data to Pyomo for an abstract model. Here is data (saved in `AbstractH.dat`) that defines a suitable happiness objective for one of the authors of this book:

```
# Pyomo data file for AbstractH.py
set A := I_C_Scoops Peanuts ;
param h := I_C_Scoops 1 Peanuts 0.1 ;
param d :=
  I_C_Scoops 5
  Peanuts 27 ;
param c := I_C_Scoops 3.14 Peanuts 0.2718 ;
param b := 12 ;
param u := I_C_Scoops 100 Peanuts 40.6 ;
```

This is a Pyomo data file, which includes `set` and `param` commands that closely resemble AMPL data commands.

NOTE: The backslash character at the end of a line tells Python that the line continues; we use it to help make the lines fit on a book page. In this particular case it is not strictly required because the line is breaking inside a parenthetical grouping.

2.4.2 A Concrete Formulation

A *concrete* Pyomo model initializes components as they are constructed. This allows modelers to easily make use of native Python data structures when defining a model instance. There are many ways to implement our model as a concrete model, and we consider one that uses Python lists and dictionaries.

```
# ConcreteH.py - Implement a particular instance of (H)
from pyomo.environ import *

model = ConcreteModel(name = "(H)")

A = ['I_C_Scoops', 'Peanuts']
h = {'I_C_Scoops': 1, 'Peanuts': 0.1}
d = {'I_C_Scoops': 5, 'Peanuts': 27}
c = {'I_C_Scoops': 3.14, 'Peanuts': 0.2718}
b = 12
u = {'I_C_Scoops': 100, 'Peanuts': 40.6}

def x_bounds(m, i):
    return (0,u[i])
model.x = Var(A, bounds=x_bounds)

def obj_rule(model):
    return sum(h[i] * (model.x[i] - (model.x[i]/d[i])**2)
               for i in A)
model.z = Objective(rule=obj_rule, sense=maximize)

model.budgetconstr = \
    Constraint(expr = sum(c[i]*model.x[i] for i in A) <= b)
```

Note that in the `budget_constr`, we define the constraint directly with the `expr` keyword argument, however, a construction rule could also be used. Also note that the lines,

```
A = ['I_C_Scoops', 'Peanuts']
h = {'I_C_Scoops': 1, 'Peanuts': 0.1}
d = {'I_C_Scoops': 5, 'Peanuts': 27}
c = {'I_C_Scoops': 3.14, 'Peanuts': 0.2718}
b = 12
u = {'I_C_Scoops': 100, 'Peanuts': 40.6}
```

could have been placed in a separate Python file and loaded with an `import` command. For example, if the data was in `mydata.py`, we could have placed the following line near the top of the file

```
from mydata import *
```

More details concerning `AbstractModel` and `ConcreteModel` are given in the next two chapters.

2.4.3 Linear Version

If we want to modify the abstract model given on page 22 to use expression (2.1), we would change the objective function expression rule as follows:

```
# AbstractHLinear.py - A simple linear version of (H)
from pyomo.environ import *

model = AbstractModel(name="Simple Linear (H)")

model.A = Set()

model.h = Param(model.A)
model.d = Param(model.A)
model.c = Param(model.A)
model.b = Param()
model.u = Param(model.A)

def xbounds_rule(model, i):
    return (0, model.u[i])
model.x = Var(model.A, bounds=xbounds_rule)

def obj_rule(model):
    return sum(model.h[i] * \
               (1 - model.u[i]/model.d[i]**2) * model.x[i] \
               for i in model.A)
model.z = Objective(rule=obj_rule, sense=maximize)

def budget_rule(model):
    return summation(model.c, model.x) <= model.b
model.budgetconstr = Constraint(rule=budget_rule)
```

Similarly, the modified concrete Pyomo model uses the same expression, as shown here:

```
# ConcreteHLinear.py - Linear (H)
from pyomo.environ import *

model = ConcreteModel(name="Linear (H)")

A = ['I_C_Scoops', 'Peanuts']
h = {'I_C_Scoops': 1, 'Peanuts': 0.1}
d = {'I_C_Scoops': 5, 'Peanuts': 27}
c = {'I_C_Scoops': 3.14, 'Peanuts': 0.2718}
b = 12
u = {'I_C_Scoops': 100, 'Peanuts': 40.6}

def x_bounds(m, i):
    return (0,u[i])
model.x = Var(A, bounds=x_bounds)

def obj_rule(model):
    return sum(h[i]*(1 - u[i]/d[i]**2) * model.x[i] \
               for i in A)
model.z = Objective(rule=obj_rule, sense=maximize)

model.budgetconstr = \
    Constraint(expr = sum(c[i]*model.x[i] for i in A) <= b)
```

2.5 Solving the Pyomo Model

Pyomo provides automated methods to (1) combine the model and data, (2) send the resulting *model instance* to a solver, and (3) recover the results for display and further use. Pyomo does not, itself, solve optimization problem instances. They are always passed to a solver of some sort.

2.5.1 Solvers

Pyomo can be installed without any solvers. For example, Pyomo can simply write out problem instances into files that are suitable as direct input to a solver. This use of Pyomo might be necessary if the solver is run separately on a different computer. Typically, however, a solver should be installed and accessible to Pyomo, and most of the examples in this book make this assumption.

Recall that the objective in (H) is not a linear function of the variable, x, and that the budget constraint is linear. Although many solvers can solve an instance with a quadratic objective and linear constraints, some solvers cannot. If the only solver on your computer is limited to linear problems, then you would need to approximate (H) with a linear model.

2.5.2 The `pyomo` Command

The `pyomo` command provides a command-line interface for solving Pyomo models. To use the `pyomo` command, the user must be in a terminal window, but **not** in a Python interpreter. To verify that pyomo is properly installed, run the command

```
pyomo --version
```

The version number of Pyomo should be displayed without error messages. The command

```
pyomo --help
```

displays some high level help on the terminal. Note that there are two dashes in `--version` and `--help`.

We assume that you have the solver `glpk` properly installed, which can be verified by running the command

```
glpsol --help
```

in your terminal. If the model file `ConcreteHLinear.py` is in the current directory, then the following command will run glpk to find a solution to the problem:

```
pyomo solve --solver=glpk ConcreteHLinear.py
```

If you have some other solver that you want to use, substitute its name for `glpk`. Note that there are two dashes in `--solver`. The `solve` subcommand displays information about the time required for each step in the optimization process as well as a little bit of information about the final solution. Information about the values of the decisions variables is put in the file `results.json` (or in `results.yml` if the Python `pyyaml` package is installed).

A data file is specified to use the `solve` subcommand with an abstract model:

```
pyomo solve --solver=glpk AbstractHLinear.py AbstractH.dat
```

Many other solvers can be used with Pyomo. For example, if the Gurobi solver is installed, then the following command can be used to solve the original, quadratic implementation of (H):

```
pyomo solve --solver=gurobi AbstractH.py AbstractH.dat
```

Also note that the solution can be printed to the screen with the `--summary` option

```
pyomo solve --solver=glpk --summary ConcreteHLinear.py
```

2.5.3 Python Scripts

An alternative to using the `pyomo` command is to add additional Python commands in the model file to explicitly optimize the model. Such a Python script is then executed using Python from the command line or within a development environment like Spyder. A script defining a concrete model can easily be solved by adding the following lines to the bottom:

```
opt = SolverFactory('glpk')

results = opt.solve(instance) # solves and updates instance

instance.display()
```

If the resulting file is called `ConcHLinScript.py`, then it can be run from the terminal with the command line:

```
python ConcHLinScript.py
```

Similarly, the following lines can be used to optimize an abstract model:

```
opt = SolverFactory('glpk')

instance = model.create_instance("AbstractH.dat")
results = opt.solve(instance) # solves and updates instance

instance.display()
```

For abstract models that rely on external data, the call to the `create_instance` method must specify the data source, as in

```
instance = model.create_instance("AbstractH.dat")
```

http://www.springer.com/978-3-319-58819-3

Pyomo — Optimization Modeling in Python

Hart, W.E.; Laird, C.D.; Watson, J.-P.; Woodruff, D.L.;
Hackebeil, G.A.; Nicholson, B.L.; Siirola, J.D.

2017, XVIII, 277 p. 13 illus., 8 illus. in color., Hardcover
ISBN: 978-3-319-58819-3