

The Diet Problem

A classic linear programming problem is the diet problem. The basis of this problem is to make sure that a person gets all of the nutrients they require at a minimum of cost. However, there are often other factors to take into consideration. Most obviously, each food has costs a different amount and has different amounts of nutrients. Additionally, ingesting too much of certain nutrients can lead to complications—you need a minimum amount of calories but you don't want to consume too many, for example. Similarly, people can only eat a specific volume of food. Fortunately, all of these constraints are easy to work with in Coopr as this example will show.

Build the model

We begin our Coopr model with

```
from coopr.pyomo import *
```

to import the Coopr package for use in the code. The next step is to create an object of the model class, which by convention we call `model`. We implement this by writing

```
model = Model()
```

The rest of our work will be contained within this object.

In the diet problem, there are two main sets that we are looking at: the set of foods and the set of nutrients, which we define as

```
model.foods = Set()
model.nutrients = Set()
```

Both of these sets could be very large (arguably, the set of all foods is unbounded); however, for implementing the model in Coopr, this is irrelevant. It is in the data file that the information for these sets is filled in.

The solution to the diet problem involves minimizing the amount of money we are spending on food. Obviously, to do this we must know the cost of each kind of food. To do this, we create a one dimensional parameter that measures the cost associated with each food:

```
model.costs = Param(model.foods)
```

This says that for each food, there is a cost associated with it, which will be implemented later when we construct the data file.

Similarly, each nutrient has a minimum and maximum requirement (note that for some nutrients the maximum is arbitrarily large). We can emulate the code for cost to create two one dimensional variables, each over nutrients.

```
model.min_nutrient=Param(model.nutrients)
model.max_nutrient=Param(model.nutrients)
```

For this model, we also want to consider how much food a person can reasonably consume—the solution is useless if no person can eat that much. Thus, we add a volume parameter over the set of foods. This is done similarly to the previous examples.

```
model.volumes=Param(model.foods)
```

We also need a parameter for the max volume of food that can be consumed. We do this like the previous examples but without a set that the parameter is over. This is because the max volume of food consumed is independent of either food or nutrients. We input

```
model.max_volume=Param()
```

There's one last parameter that must be taken into consideration: how much of each nutrient each food contains. Unlike the previous examples, this parameter is in two dimensions—it is over both the set of foods and the set of nutrients. This just requires a small change in the code.

```
model.nutrient_value=Param(model.nutrients, model.foods)
```

The comma indicates that this parameter is over two different sets, and thus is in two dimensions. When we create the data file, we will be able to fill in how much of each nutrient each food contains.

At this point we have defined our sets and parameters. However, we have yet to consider the amount of food to be bought and eaten. This is the variable we're trying to solve for, and thus we create an object of the variable class. Since this is just recording how much food to purchase, we create a one dimensional variable over food:

```
model.amount=Var(model.foods, within = NonNegativeReals)
```

We restrict our domain to the non-negative reals. If we accepted negative numbers then the model could tell us to buy negative amounts of food, which is an unrealistic—and thus useless—model. We could further restrict the domain to the integers to make it more realistic, but that would make the problem much harder for little gain: if this model is used on a large scale then the difference between the integer solution and the non-integer solution is often irrelevant.

At this point we must start defining the rules associated with our parameters and variables. We begin with the most important rule, the cost rule, which will tell the model to try and minimize the overall cost. Logically, the total cost is going to be the sum of how much is spent on each food, and that value in turn is going to be determined by the cost of the food and how much of it is purchased. For example, if three \$5 hamburgers and two \$1 apples are purchased, then the total cost would be $3 \cdot 5 + 2 \cdot 1 = 17$. Note that this process is the same as taking the dot product of the amounts vector and the costs vector.

To input this, we must define the cost rule, which we creatively call `costRule` as

```
def costRule(model):
    return sum(model.costs[n]*model.amount[n] for n in model.foods)
```

which will go through and multiply the costs and amounts of each food together and then take their sum as outlined above. We must include another line, though:

```
model.cost=Objective(rule=costRule)
```

This line defines the objective of the model as the costRule, which Coopr interprets as the value it needs to minimize; in this case it will minimize our costs. Also, as a note, we defined the objective as “model.cost” which is not to be confused with the parameter we defined earlier as “model.costs,” despite their similar names. These are two different values and accidentally giving them the same name will cause problems when trying to solve the problem.

We must also create a rule for the volume consumed. The construction of this rule is similar to the cost rule as once again we take the dot product, this time between the volume and amount vectors.

```
def volumeRule(model):
    return sum(model.volumes[n]*model.amount[n] for n in
               model.foods) <= model.max_volume
```

```
model.volume = Constraint(rule=volumeRule)
```

Note that here we have a constraint instead of an objective. This requires that the rule returns true, but otherwise the value is irrelevant. While objective looks for the least value, constraints just require that a value works.

Finally, we need to add the constraint that ensures we obtain proper amounts of each nutrient. This one is a bit more complicated for two reasons: the value needs to be within a range, rather than just greater than or less than another value, and nutrient_value was a two dimensional variable. It's easy to fix the first problem in a myriad of ways; the way we will do it involves defining another variable and checking if that is in the proper range. To solve the second problem, we give the rule an index in addition to the model as an input. The code will be

```
def nutrientRule(n, model):
    value = sum(model.nutrient_value[n,f]*model.amount[f]
                for f in model.foods)
    return (model.min_nutrient[n], value, model.max_nutrient[n])
```

```
model.nutrientConstraint = Constraint(model.nutrients, rule=nutrientRule)
```

The rule itself will act much like the previous rules, but by adding an index into the constraint we will cycle through each of the nutrients. Essentially, what we have done is compressed many “nutrient rules,” each of which acts the same, into one rule that will look at each nutrient individually.

At this point, we have finished creating the model file. We have defined our sets, parameters and variables. We also defined the objective of the model and constraints that must be accepted for a solution. All that's left now is to build a data file.

Data entry

Much like with the model, we begin with the two main sets we're looking at: foods and nutrients. For brevity, we'll only look at three foods and three nutrients. Note that "vc" is just shorthand for vitamin c.

```
set foods := steak apple rice;
set nutrients := calories protein vc;
```

To define the set just put set [name] := [elements of set]; where the elements of the set are separated by a single space.

We now define a parameter without an associated set. In this case, it is the parameter "max_volume". To do this, simply input

```
param: max_volume := 400;
```

It is worth pointing out that for this example the volumes are all fairly arbitrary.

The parameters representing the costs, volumes, nutrient minimums and nutrient maximums can all be input in comparable fashions with the main difference being which set the parameter is over. In the code, the first line defines what parameter is being looked at and each subsequent line gives a member of the appropriate set and a value associated with it.

```
param: costs :=
steak 10
apple 2
rice 1;
```

```
param: volumes :=
steak 1
apple 1
rice 1;
```

```
param: min_nutrient :=
calories 2000
protein 56
vc 300;
```

```
param: max_nutrient :=
calories 4000
protein 168
vc 2000;
```

For the most part, these numbers are arbitrary and should not be used for any real life diet. One note is that the reason all of the volumes are the same is because we are standardizing all of the prices (and soon nutrients) per 1 unit of volume. In a different example, such as comparing foods at a restaurant, the volumes would be different among the foods, but in this case they're the same.

Finally, we create the data for our parameter `nutreint_value`, which was over both food and nutrients. Once again, we include “param” followed by the name of the parameter on the first line (though note that there is no colon after param in this case, while there was previously). The next line is the elements of the food set followed by `:=` and the first column is the elements of the nutrient set. We then fill in the resulting matrix.

```
param nutrient_value:
      steak apple rice :=
calories    180   65   143
protein     40    1    5
vc          0    30    0;
```

The amount of spaces between each element is irrelevant (as long as there is at least one) so the matrix should be formatted for ease of reading.

Now that we have finished both the model and the data file save them both. It's convention to give the model file a `.py` extension and the data file a `.dat` extension.

Solution

Using `Coopr` we quickly find the solution to our diet problem: purchase 9.44 units of volume of rice and 10 apples. Of course, further refinements could be made, such as requiring at least one of each kind of food, a maximum amount purchased of one food, a “happiness” factor that takes into account how much you enjoy each food, or just including more foods and nutrients. However, this is left as an exercise for the reader—with this outline it should be simple to refine the model for more complicated scenarios.