BEE 4750 Lab 3: Linear Programming with JuMP

2024-08-05

> Due Date
>
> Wednesday, 10/16/24, 9:00pm

Setup

The following code should go at the top of most Julia scripts; it will load the local package environment and install any needed packages. You will see this often and shouldn't need to touch it.

```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
using JuMP # optimization modeling syntax
using HiGHS # optimization solver
using Plots # plotting
```

Overview

In this lab, you will write and solve a resource allocation example using `JuMP.jl`. `JuMP.jl` provides an intuitive syntax for writing, solving, and querying optimization problems.

For an example of using `JuMP.jl` to solve linear programs, see the relevant tutorial on the class website.

Free free to delete some of the illustrative cells and code blocks in your notebook as you go through and solve the lab problems…this might help reduce some potential confusion while grading about what your answer is.

`JuMP.jl` Syntax

Feel free to consult the website's JuMP tutorial for syntax help. The keys:

1. Initialize your model with a solver; in this case, we'll use the `HiGHS` solver, but there are other solvers listed here for different types of problems, some of which are open and some of which require a commercial license: https://jump.dev/JuMP.jl/stable/installation/#Supported-solvers:

```
example_model = Model(HiGHS.Optimizer)
```

1. Define variables with syntax like

   ```
   #| output: false
   @variable(example_model, 1 >= example_x >= 0)
   ```

   This will create a variable `example_x` which is constrained to be between 0 and 1; you can leave off any of the bounds if a variable is unbounded in a particular direction. You can also add a vector of variables:

   ```
   #| output: false
   T = 1:3 # define set to index variables
   @variable(example_model, 1 >= example_z[t in T] >= 0)
   ```

   which will create a vector of 3 variables `example_z[1]`, …, `example_z[3]`, all of which are bounded between 0 and 1.

2. Add an objective with

   ```
   #| output: false
   @objective(example_model, Max, example_x + sum(example_z))
   ```

   which will add an objective to maximize (replace with `Min` to minimize).

3. Add constraints:

   ```
   #| output: false
   @constraint(example_model, constraint1, 2example_x + 3*sum(example_z) <=
    ↪  10)
   @constraint(example_model, constraint2, 5example_x - example_z[1] <= 2)
   ```

which will name the constraints `constraint1` and `constraint2` (you should make yours more descriptive about what the constraint actually is). The value of adding a name is to facilitate later querying of shadow prices, which we will discuss later. You can also add a vector of constraints which have similar structure or rely on different elements of a data vector:

```
#| output: false
A = [2; 4]
b = [8; 12]
I = 1:2 # set indices for constraint
@constraint(example_model, vector_constraint[i in I], A[i] *
 ↪  sum(example_z) .<= b[i])
```

You can also define matrices of constraints which depend on two index sets by generalizing this syntax, e.g.

```
@constraint(example_model, matrix_constraint[i in I, j in J, ...])
```

> **Tip**
>
> Specifying higher-dimensional vectors and matrices of variables and constraints will be important when we start looking at more complex applications, so don't skip over this! You don't want to manually enter thousands of constraints to ensure hourly electricity demand is met…

Finally, you can (and probably should) `print` your model to make sure that you get something that looks like the equations that you wrote down (in a notebook, this will be nicely rendered):

```
print(example_model)
```

> **Define your entire model in one cell**
>
> `JuMP` has great and intuitive syntax (this is one of the reasons we're using Julia in this course!), but it doesn't like re-defining variables or constraints once they've been set. I recommend putting all of your model-definition code (starting from `model = Model(...)`) for a particular optimization problem in a single notebook cell, so you can re-set up the entire problem with a single click when you want to make a change.

Instructions

Remember to:

- Evaluate all of your code cells, in order (using a `Run All` command). This will make sure all output is visible and that the code cells were evaluated in the correct order.
- Tag each of the problems when you submit to Gradescope; a 10% penalty will be deducted if this is not done.

Exercise (3 points)

Your task is to decide how much lumber to produce to maximize profit from wood sales. You can purchase wood from a managed forest, which consists of spruce (320,000 bf) and fir (720,000 bf). Spruce costs \$0.12 per bf to purchase and fir costs \$0.08 per bf.

At the lumber mill, wood can be turned into plywood of various grades (see Table 1 for how much wood of each type is required for and the revenue from each grade). Any excess wood is sent to be recycled into particle board, which yields no revenue for the mill.

Table 1: Wood inputs and revenue by plywood grade. S refers to spruce inputs, F fir inputs.

Plywood Grade	Inputs (bf/bf plywood)	Revenue (\$/1000 bf)
1	0.5 (S) + 1.5 (F)	400
2	1.0 (S) + 2.0 (F)	520
3	1.5 (S) + 2.0 (F)	700

You will go through the steps of formulating and solving a linear program for this problem.

Make sure to:

- Clearly define your decision variables notation, including what variables you are using, what they mean, and what their units are.

  > **Selecting Decision Variables**
  >
  > There isn't always a single obvious choice for your decision variables, but some make the rest of your problem formulation easier than others. You can always go back and re-assess if you struggle with defining your objective function and constraints with a particular choice!

- Derive your objective function. Support your function with justifications and/or equations as necessary.
- Derive any needed constraints. Support your function with justifications and/or equations as necessary.

- Put this optimization problem in mathematical programming form. For an example of the syntax for this in a notebook (or other LaTeX-based formatting), see lines 82–91 here.

Extra:

You may not have time to complete this part of the lab before submitting, but since the problem is relatively simple, these steps might be worth returning to as you work on the linear programming homework.

- Code your linear program using `JuMP`.
- Find the solution to your program and find the optimal values of the decision variables. Once you've defined your model, you can find the solution with `optimize!()`:

```
optimize!(example_model)
```

> **What if I Get An Error?**
>
> If `optimize!()` throws an error, that's usually a sign that something is wrong with the formulation (for example, a variable might not be bounded or a constraint might not be specified correctly) or a typo in the model definition. Linear programs should be well behaved!

To find the values of variables after optimizing, use `value.()` (the broadcasting ensures this will work for vector-valued variables as well):

```
value.(example_x)
```

```
value.(example_z)
```

- Find the shadow price of the spruce constraint; remember that this is the change to profit from every additional unit of spruce that becomes available. You can get the shadow price of a constraint with the following syntax:

```
shadow_price(constraint1) # this is why we named the constraints when we
 ↪  defined them
```

References

Put any consulted sources here, including classmates you worked with/who helped you.