BEE 4750 Lab 3: Linear Programming with JuMP

Name: Anthony Nicolaides

ID: ajn68

> **Due Date**
>
> Friday, 10/13/23, 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.

In [ ]:
```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
  Activating project at `~/Documents/BEE4750/labs/lab03-anthonynic28`
```

In [ ]:
```
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.

Introduction

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. This resource allocation problem is diagrammed in Figure 1.

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

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

Figure 1: Flowchat of the resource allocation problem in this lab.

Problems (10 points)

Problem 1: Problem Formulation (5 points)

In this problem, you will go through the steps of formulating a linear program for this problem.

Problem 1.1 (1 point)

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

Variable	Meaning	Units
P1	the amount of grade 1 plywood made from spruce and fir	1000 bf
P2	the amount of grade 2 plywood made from spruce and fir	1000 bf
P3	the amount of grade 3 plywood made from spruce and fir	1000 bf

Problem 1.2 (1 point)

Derive your objective function. Support your function with justifications and/or equations as necessary. You will not receive credit just for the function alone.

Objective function is the max value between the profits derived from each plywood grade, so profit = revenue - cost:

Note: All units are in 1000 bf

Revenue for each grade is simply:

```
P1 * 400

P2 * 520

P3 * 700

revenue = (P1 * 400) + (P2 * 520) + (P3 * 700)
```

Cost for each grade is a bit more involved, so we know the total cost is the amount of spruce and amount of fir purchased, so:

```
120 * S + 80 * F
```

However, we need to segment this to find the cost per plywood grade, here we utilize the inputs from Table 1:

```
P1: 0.5S + 1.5F

P2: 1.0S + 2.0F

P3: 1.5S + 2.0F
```

So, to find how much spruce and fir is used for each plywood, we simply multiple these ratios by the amount of plywood that was made:

```
Spruce used for plywood grade 1: 0.5*P1

Spruce used for plywood grade 2: 1.0*P2

Spruce used for plywood grade 3: 1.5*P3


Fir used for plywood grade 1: 1.5*P1

Fir used for plywood grade 2: 2.0*P2

Fir used for plywood grade 3: 2.0*P3
```

This lets us put the cost in terms of our decision variables, P1, P2, and P3.

Now, we return to the equation that gives the total cost of spruce and fir purchased:

```
S = 0.5 * P1 + 1.0 * P2 + 1.5 * P3

F = 1.5 * P1 + 2.0 * P2 + 2.0 * P3

120 * S + 80 * F

cost = 120 * (0.5 * P1 + 1.0 * P2 + 1.5 * P3) + 80 * ( 1.5 *
P1 + 2.0 * P2 + 2.0 * P3)
```

Now, we find the total profit, in terms of our decision variables:

```
(P1 * 400) + (P2 * 520) + (P3 * 700) - 120 * (0.5 * P1 + 1.0
* P2 + 1.5 * P3) - 80 * ( 1.5 * P1 + 2.0 * P2 + 2.0 * P3)
```

Rearranging it gives:

```
(400 * P1 - 0.5 * 120 * P1 - 1.5 * 80 * P1) +
(520 * P2 - 1.0 * 120 * P2 - 2.0 * 80 * P2) +
(700 * P3 - 1.5 * 120 * P3 - 2.0 * 80 * P3)
```

This is our objective function.

Problem 1.3 (2 point)

Derive any needed constraints. Support your function with justifications and/or equations as necessary. You will not receive credit just for the final constraints alone.

There is a limit to how much spruce and fir that can be purchased:

```
0 <= S <= 320(units:1000bf)

0 <= F <= 720(units:1000bf)
```

And since we know spruce and fir purchased can be expressed in terms of P1, P2, and P3:

```
S = 0.5 * P1 + 1.0 * P2 + 1.5 * P3

F = 1.5 * P1 + 2.0 * P2 + 2.0 * P3
```

Also, you can not make negative plywood, therefore, our constraints are:

```
0 <= 0.5 * P1 + 1.0 * P2 + 1.5 * P3 <= 320

0 <= 1.5 * P1 + 2.0 * P2 + 2.0 * P3 <= 720

0 <= P1

0 <= P2

0 <= P3
```

Problem 1.4 (1 point)

Put this optimization problem in mathematical programming form. For an example of the syntax for this, see lines 82–91 here.

$$
\begin{aligned}
\max_{P1,P2,P3} \quad & (400 * P1 - 0.5 * 120 * P1 - 1.5 * 80 * P1) + \\
& (520 * P2 - 1.0 * 120 * P2 - 2.0 * 80 * P2) + \\
& (700 * P3 - 1.5 * 120 * P3 - 2.0 * 80 * P3) \\
\text{subject to} \quad & \\
& (0.5 * P1 + 1.0 * P2 + 1.5 * P3) \geq 0 \\
& (0.5 * P1 + 1.0 * P2 + 1.5 * P3) \leq 320 \\
& (1.5 * P1 + 2.0 * P2 + 2.0 * P3) \geq 0 \\
& (1.5 * P1 + 2.0 * P2 + 2.0 * P3) \leq 720 \\
& P1 \geq 0 \\
& P2 \geq 0 \\
& P3 \geq 0
\end{aligned}
\tag{1}
$$

Problem 2: Find the Solution (5 points)

Problem 2.1 (2 points)

Code your linear program using `JuMP`. 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)
   ```
2. Define variables with syntax like

```
@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:

```
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.

3. Add an objective with

```
@objective(example_model, Max, example_x + sum(example_z))
```
which will add an objective to maximize (replace with `Min` to minimize).

4. Add constraints:

```
@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:

```
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)
```

$$
\begin{aligned}
\max \quad & example\_x + example\_z_1 + example\_z_2 + example\_z_3 \\
\text{Subject to} \quad & 2example\_x + 3example\_z_1 + 3example\_z_2 + 3example\_z_3 \leq 10 \\
& 5example\_x - example\_z_1 \leq 2 \\
& 2example\_z_1 + 2example\_z_2 + 2example\_z_3 \leq 8 \\
& 4example\_z_1 + 4example\_z_2 + 4example\_z_3 \leq 12 \\
& example\_x \geq 0 \\
& example\_z_1 \geq 0 \\
& example\_z_2 \geq 0 \\
& example\_z_3 \geq 0 \\
& example\_x \leq 1 \\
& example\_z_1 \leq 1 \\
& example\_z_2 \leq 1 \\
& example\_z_3 \leq 1
\end{aligned}
$$

> **Define your entire model in one cell**
>
> `JuMP` has great and intuitive syntax, 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.

In [ ]:
```
plywood_model = Model(HiGHS.Optimizer)

@variable(plywood_model, P1 >= 0)
@variable(plywood_model, P2 >= 0)
@variable(plywood_model, P3 >= 0)

@constraint(plywood_model, constraintS,
    0.5 * P1 + 1 * P2 + 1.5 * P3 <= 320)

@constraint(plywood_model, constraintF,
    1.5 * P1 + 2.0 * P2 + 2.0 * P3 <= 720)

@objective(plywood_model, Max,
    400 * P1 - 0.5 * 120 * P1 - 1.5 * 80 * P1 +
    520 * P2 - 1.0 * 120 * P2 - 2.0 * 80 * P2 +
    700 * P3 - 1.5 * 120 * P3 - 2.0 * 80 * P3)

println(plywood_model)
```

```
Max 220 P1 + 240 P2 + 360 P3
Subject to
 constraintS : 0.5 P1 + P2 + 1.5 P3 ≤ 320
 constraintF : 1.5 P1 + 2 P2 + 2 P3 ≤ 720
 P1 ≥ 0
 P2 ≥ 0
 P3 ≥ 0
```

Problem 2.2 (1 points)

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!():

In [ ]:
```
optimize!(plywood_model)
```

```
Running HiGHS 1.5.3 [date: 1970-01-01, git hash: 45a127b78]
Copyright (c) 2023 HiGHS under MIT licence terms
Presolving model
2 rows, 3 cols, 6 nonzeros
2 rows, 3 cols, 6 nonzeros
Presolve : Reductions: rows 2(-0); columns 3(-0); elements 6(-0) - Not reduce
d
Problem not reduced by presolve: solving the LP
Using EKK dual simplex solver - serial
  Iteration        Objective     Infeasibilities num(sum)
          0    -8.1999929744e+02 Ph1: 2(8.5); Du: 3(819.999) 0s
          2     1.1200000000e+05 Pr: 0(0) 0s
Model   status      : Optimal
Simplex   iterations: 2
Objective value     :  1.1200000000e+05
HiGHS run time      :          0.00
```

> **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):

In [ ]:
```
# round to avoid float value errors:
#   P1 = 352.00000000000006
#   P2 = 0.0
#   P3 = 95.99999999999999
println("Plywood Grade 1: ", round(value.(P1)), " 1000bf")
println("Plywood Grade 2: ", round(value.(P2)), " 1000bf")
println("Plywood Grade 3: ", round(value.(P3)), " 1000bf")
println("Profit: \$", round(objective_value(plywood_model)))
```

```
Plywood Grade 1: 352.0 1000bf
Plywood Grade 2: 0.0 1000bf
Plywood Grade 3: 96.0 1000bf
Profit: $112000.0
```

Problem 2.3 (1 point)

How would your profit change if you could buy 1,000 additional bf of spruce? You can answer this by getting the shadow price of a particular variable with:

In [ ]:
```
# this is why we named the constraints when we defined them
shadowPriceS = shadow_price(constraintS)
println("Profit would increase by \$", shadowPriceS,
    " if 1000 additional bf of spruce could be purchased")
```

```
Profit would increase by $80.0 if 1000 additional bf of spruce could be purch
ased
```

Problem 2.4 (1 point)

Would you prefer to have 2,000 additional bf of spruce or 1,000 additional bf of fir?

In [ ]:
```
shadowPriceF = shadow_price(constraintF)

# shadow price shows how profit would change in constraint was realxed by
#   one unit (1 unit = 1000 bf), therefore 2000 additinoal bf is double
#   the shadow price
println("Profit increase if purchased 2000 additional bf of spruce: \$",
    2 * shadowPriceS)
println("Profit increase if purchased 1000 additional bf of fir: \$",
    shadowPriceF)

println("\nI would prefer to purchase 2000 additional bf of spruce, as it
    gives \$40 more profit than if I purchased 1000 additional bf of fir")
```

```
Profit increase if purchased 2000 additional bf of spruce: $160.0
Profit increase if purchased 1000 additional bf of fir: $120.0

I would prefer to purchase 2000 additional bf of spruce, as it
    gives $40 more profit than if I purchased 1000 additional bf of fir
```

References

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

Consulted TA Gabriela Ackermann Logan and Profressor Vivek Srikrishnan