Practicals for Data Analysis & Programming for Operations Management

N.D. Van Foreest, W. van Wezel

1. INTRODUCTION

This document contains the material for the practicals related to optimization of Operations Management problems with Python, Gurobi, and Elasticsearch. We rely on you to search on web for examples, documentation, and so on. For instance, a search on Python lp examples will provide you with lots of useful websites that provide code examples on how to use python to solve LP problems. In fact, by reading (thoroughly!) code examples of others, we learned to program in Python ourselves. First we looked for simple examples, then we read harder examples, and so on.

At the end of each practical week we will make our code available, so that you can compare your solution with ours. Given that students will share their code anyway via, e.g., Dropbox anyway, there is no point in not giving you the code. In fact, by giving you our code, we can at least ensure you get good code. However, please try real hard to complete the practicals yourself.

2. PRACTICAL 1B: PRODUCT-MIX OPTIMIZATION

2.1 Get a working python environment with Gurobi

For the practicals, you will have to install a working Python environment and the mathematical optimization solver from Gurobi, which is perhaps the best LP optimization library available, used by nearly every multi-national with optimization problems such as KLM, Schiphol, IBM, and so on). See https://www.gurobi.com/. How to get the Python environment is explained in the manual for the a-practicals (generic programming skills). So here, we assume you have a Python and Atom installed on your computer. To install Gurobi and link it to Python, follow these steps:

1. Download Gurobi on https://www.gurobi.com/. On the top of the screen, click 'downloads & licenses', and then click on 'download center'.
2. In the download center, click on 'Gurobi Optimizer'
3. Accept the license
4. Then download the appropriate version of the latest release of the Gurobi Optimizer (this depends whether you are on Windows, MacOs, or linux).
5. You can then install Gurobi.
6. Now go back to the download center, and, in the 'request a license' section, click on 'Academic License'.
7. You need to register an account. Make sure to use your university email address; they might require this to verify you are indeed a student from academia.
8. When you have registered, you can generate a license code.
9. When generated, you can activate this license code on your computer:
    a. Open a command prompt (Windows: click the windows button, type cmd, and press enter; on MacOs: open a terminal).
    b. Copy the command provided by the Gurobi website and press enter. The command should look like this (the numbers will be different for you):
    `grbgetkey 123456-abcde-321232-6456-2342432432`

If you don't get error messages, Gurobi is now installed. We now must arrange for Python to be able to use Gurobi. This requires the installation of the Gurobi Python libraries. The easiest way to do this is start a command prompt or terminal again, and go to the directory where Gurobi is installed. For me this was C:\gurobi903\win64. So after opening the command prompt, I type `cd C:\gurobi903\win64`, and then press enter. This folder should contain a file called `setup.py`. On your command prompt, now type `python setup.py install`. This will install the Gurobi libraries in Python.

2.2 Running the product mix code

1. Download product_mix.py from nestor; the code is also available in Table 1. This code implements a linear program (LP), see Factory Physics Ch 16 for further explanation.
2. Run this code in Atom.
3. Tip: if you run the code and you get the output in a small popup window, then right-click on the code window, click 'hydrogen' in the menu, and then click ''toggle output area'. Now run the code again. The output will now be in the right window.
4. Read and think about the output.

2.3 Breaking the code

It is extremely useful to become familiar with python's error message. When you are coding, you'll often make mistakes (this is entirely normal; it happens to us all the time). Python's error messages can be enormously helpful in discovering what you did wrong. For this reason, we are going to break our working code, and see what type of errors we get. Read the error trace carefully; for instance, check the line number where the error occurs, and check the error type.

Here are some suggestions to break the code. Do them all, and then try your own mistakes. Read the error message very carefully. Understanding the error messages can save you an enormous amount of time (not just minutes or hours, but days!) After introducing the error, don't forget to repair, i.e., undo, it.

1. Comment Line 10, see Table 1, in your code; that is, put a # in front of the line. (Mind that python is sensitive to indentation, hence, the # should be the very first character on the line, there should be NOTHING in front of it). What is the error message? Remove the comment sign again.
2. Now comment out Line 12 (and uncomment it after having read the error message).
3. Line 12, put a space in front of the first word, so that everything moves one step to the right. Check the error message, and explain it.
4. Line 12, what would happen if you would put two spaces at the front of the line?
5. Line 12, what would happen if you would put a tab character at the front of the line?
6. Comment out Line 14.
7. In Line 14, change addVar to addvar. (Just change the capital 'V' to 'v').
8. In Line 14, change x1 to x.
9. Comment out Line 17.
10. Comment out Line 19.
11. Comment out Line 21.
12. Line 21, remove the number 2400 so that the line reads like: `m.addConstr(15 * x1 + 10 * x2 <= )`
13. Comment out Line 26.
14. Comment out Line 29. What happens? Explain it.
15. Line 26: replace the line by `for v in:`
16. Introduce your own errors, one by one, and repair again. The more errors you can invent, the better. Once again, reading and understanding python's error messages will save you lots and lots of time, and frustration.

Table 1: product_mix.py

```python
#!/usr/bin/python

# This example  implements the product mix example of FP,
# Appendix 16.A. See eqs 16.107--16.113.

# Nicky van Foreest, 2019

%reset -f

from gurobipy import Model, GRB

m = Model("product mix")

x1 = m.addVar(ub=100, name="x1")
x2 = m.addVar(ub=50, name="x2")

m.setObjective(45 * x1 + 60 * x2, GRB.MAXIMIZE)

m.addConstr(15 * x1 + 10 * x2 <= 2400)
m.addConstr(15 * x1 + 35 * x2 <= 2400)
m.addConstr(15 * x1 + 5 * x2 <= 2400)
m.addConstr(25 * x1 + 14 * x2 <= 2400)

m.optimize()

for v in m.getVars():
    print("%s %g" % (v.varName, v.x))

print("Obj: %g" % m.objVal)

```

2.4 *Integer variables, get familiar with the gurobi documentation.*

In the solution you must have seen that the optimal number of products to be produces is a fraction. Check the Gurobi documentation how to enforce that x1 and x2 are integer. (Just search the web. . .) Hint, google for `gurobi addvar integer`.

You should know that optimization problems whose variables are constrained to integers are typically much, much more complicated than problems all of whose variables are continuous. If discrete variables are not necessary, do not include the GRB.INTEGER property! Including this property in the problem specification for Gurobi, i.e., including the integer property in the code, can easily change the computation time from seconds to years for industry size problem. To understand the problem, suppose that the optimal value of x1 is either 75 or 76, and for x2 it is either 35 or 36. To find the optimal integer solution we need to test 2 x 2 = 2^2 = 4 alternatives. If we have 100 decisions variables, we need to test about 2^{100} possibilities; this is a huge number, impossible to test in any reasonable amount of time. So be aware!

2.5 Finding the constraints

Ensure that your code is equal to the original code again, so that you have a correctly working example. Let Gurobi solve the problem. Then you should notice that not all demand is covered. By changing the production system it must be possible to increase sales. But. . . , which machine to change, and what to do? In the next couple of exercises we will tackle this problem.

You should observe, and remember, from these problems how easy the numerical analysis becomes with our Python/Gurobi environment. We generate some ideas to extend production capacity. To see the effect of these suggestions, we have to change some numbers in our code, but that is very easy. Then we let Gurobi solve the problem, and we read out the results, and compare the effect on the total revenue. In other words, we (as humans) suggest plans on how to improve things, we let the computer carry out all the boring computations, and then we (as humans) interpret the results.

Now, our plan is to add capacity. But which machine(s) should we update? Which is the best to invest in?

A simple (but rather dumb) way to find out the machine that is the most constraining is as follows. We remove a machine from the set of constraints in lines 16–19, and just check whether the output (i.e., the value of the objective) increases.

1. First comment Line 19, and let gurobi solve the problem. Check what happens to the solution.
2. Now include Line 19 again, but comment Line 20. Let gurobi solve the problem and check the solution.
3. And so on.

You should see that machines B and D have an impact on the revenue.

You should know that the above procedure is by far not the smartest. You can use Gurobi to identity the, so-called, active constraints. However, this requires more knowledge of Gurobi, which you can always acquire later.

2.6 Extra minutes for the constraining machines

The learning goal for this and the next session is that you understand how easy it is with computers to quantitatively analyze many different scenarios. As an example, suppose we can buy some overtime, let's say 100 minutes per week on any machine. We would like to know which machine is the best. The next couple of questions are meant to help you get started with asking and answering such questions.

1. Change the constraint on machine B to 2500. Then solve the problem again, with Gurobi. And check the result.
2. What happens if you just add 1000 minutes to machine B? Realize that some of this extra time might be unused. Then this is not-so-smart decision: we pay for personnel, but we are unable to exploit this extra capacity due to the other machines.
3. Undo the changes on machine B, but change machine D's time to 2500. Then solve the problem again, with Gurobi. And check the result.
4. Perhaps it it better to give 50 minutes to machine B and D. What is the result?
5. How many extra minutes are necessary to satisfy all demand? (You don't have to find a general procedure for this problem, just play a bit with the numbers to see the effect.)
6. Another idea is to move some of the total time of machine A to machine B. In other words, we train the personnel that operates machine A to help at machine B, thereby increasing the total available time on machine B (but reducing the time of machine A). Let's assume that the operators of machine A are less fast when they help at machine B. For instance, if we remove 200 minutes from machine A then this can be converted to just 100 minutes at machine B. Should we consider training machine A's personnel to help at machine B? Finding out is easy: change the 2400 of machine A into 2300, and increase the 2400 to 2450 at machine B. Have Gurobi solve this and interpret the result.
7. Make a graph of the output as a function of the extra capacity added to machine B. For the present, it is ok to make this in Excel. Later you'll learn how to make this with Python tools.

We assume that by now you understand how easy scenario analysis becomes. You should realize that scenario analysis is one of the key skills of managers. Generate good ideas on what to change, analyze the effects of the changes, interpret the results, and implement the best change.

2.7 Moving work from a bottleneck to a non-bottleneck

Another way to shift work from machine B to machine A (or C) is to reduce the production time. The idea is to assume that part of the work of an item can be done at either machine.

1. (Don't forget to bring your code to a pristine state again.)
2. What is the impact of increasing the production time of product 1 at machine A from 15 to 20 minutes, but thereby reducing the production time at station B from 15 to 10 minutes?
3. Should we move time for product 1 to station A or station C?
4. or should we move production time of product 2?
5. Should we move time from station D to station C? Which product?
6. How much time should we try to move? This question is particularly interesting. Suppose we would be able to redesign product 1 so that we can move all its production time at machine B to machine A, so that machine B can be bypassed altogether. Would it be useful to investigate such a design? (If such a redesign does not result in much extra output, it's useless to try. So, with our computations we can first analyze the effect on logistics and revenue before we start thinking about how to do the redesign.)

2.8 *Testing*

Finally, we need to check whether our model is correct. Note that for a real case it is essential to start with testing your model and implementation. In this practical we did the scenario first, so we sinned a bit. The reason is to show you first how to use computational tools such as Python and Gurobi, and not bore you with testing. But remember, when you use tools to make decisions for real, then always test first. It's easy to make a typo (e.g., type in the wrong number), miss a constraint, and so on, and it is a bit painful when your multi-million decision turns out not to work because you typed in the wrong numbers. . .

1. Remove all machine constraints, by commenting them out, and check that only the demands form the constraints.
2. Include the machine constraints again, but now remove the machine constraints on the demands, by removing the upper bounds indicated by the key word 'ub' in the code. (Check Gurobi's documentation on how to remove the upper bound.) Use Figure FP 16.15 to see what would be the optimal solution for this case, and check that you get the same solution with gurobi.

2.9 *Adding Machines*

Yet another idea is to buy an extra machine that can take off some of the load of machine D, say. So, we are going to add a machine E to the end of the chain of machines.

1. Copy product_mix.py to product_mix_fifth_machine.py, and use the latter file to implement the changes.
2. Add to the LP a machine E with production times 13 and 7 for products 1 and 2, respectively. Machine E is available for 1500 minutes per week. The production times at station D change now to 12 and 10 respectively.
3. Analyze the quantitative effect of moving work from machine D to machine E. (I see an increase in revenue from 5575 to 6042 by adding a machine E.)

So, based on estimates of the cost of buying and operating such a machine E we can use our computer programs to make good decisions, i.e., whether it's worth the money or not.

2.10 Adding a third product

A marketeer suggests to add a third product to our product portfolio. The estimated production times are 10, 5, 18, 10 on the respective machines; the selling price is $120 and the cost of raw material including labor is $30. The demand is estimated to be 20 products per week. Should we include this product in our portfolio?

1. Copy product_mix.py to product_mix_third_product.py.
2. Add product 3 to the LP and analyze the effect, in particular on the total profit. Does the profit increase when you add product 3?
3. Due to contractual obligations we have to minimally produce 80 items of product 1 and 20 of product 2. Include this in the LP. (tip: in `addVar`, `ub` stands for upper bound. Check the impact.

2.11 Minimizing the capacity required

Here is an interesting challenge: What would be the minimal capacity per machine required to serve all demand? You can build this also as an LP. Here I leave the details to you. The solution becomes available at the end of the week. Tip: what should be your decision variables now?

Upload your code of 2.11 to Nestor as end product of Week 1.

3 PRACTICAL 2B: CODE ORGANIZATION AND INVENTORY CONTROL

The code in Table 1 has some problems. It does not scale to large numbers of machines or products because each product and machine has to be included by 'hand'. Moreover, the data is hard-coded in the algorithm itself. It is a much better design pattern to separate data from the algorithm, and make the algorithm generic in the sense that it can scale up to many products or machines. That is what we are going to learn in Section 3.1.

In Section 3.2 we'll organize our code a bit more with functions. With this step we complete the product mix example.

Then, in Section 3.3, we will analyze an inventory control problem with LP. This is a really useful example; over the years students have been applying it at companies to control the production and inventories.

In the next practicals we will no longer take you by the hand, but just give you hints on how to work towards an interesting computer program.

3.1 Separating data from the formal problem specification

We continue extending the case of the previous practical.

1. Create a new file, product_mix_2.py, so that you don't mess up the program in product_mix.py (This one worked, so while developing something new, it is best not to break the stuff that works.)

2. Put demand data in a list, called array `D`.
3. Put profits in a list `P`.
4. Put the capacity constraints of the machines in a list `C`.
5. Put the production times in a list of lists, `PT`, such that, for instance, `PT[0][1]` corresponds to the production time of product 2 at machine A (recall that python starts counting at 0, hence we subtract 1 from the machine id and product id).
6. Next, we need to make decision variables. Search, for instance, on `gurobi multiple variables example`. Figure out how to make a list (also called array) of decision variables.
7. We now need to add the constraints. A single constraint often refers to multiple decision variables and data elements. In our case, we need to multiply the decision variable with the processing time, and compare the result to the maximum machine capacity. In the previous version of the code, we manually added a constraint for each machine. Now that the machines and products are configurable we need to add the constraints in a loop. Conceptually, we would need something like:

```
for i in range(len(C)):
        m.addConstr(time_used_on_this_machine<=C[i])
```

where the time used on the machine would be the multiplication of all products produced on that machine multiplied by the amount produced (which is our decision variable x). So, within a single constraint, we need to iterate over all products. This is where generators and the Python `sum` function can help us. For example, try the following code:

```
x = sum(2 * n for n in range(5))
print(x)
```

The code within the `sum` function is called a generator. Track that this code is a shorter version of:

```
x = 0
for n in range(5):
    x = x + 2 * n
print(x)
```

(ps, if you expected the outcome of this code to be 30, please read `https://www.w3schools.com/python/ref_func_range.asp`)

This can also be used when adding constraints. It could looks something like:

```
num_machines = len(C)
for i in range(num_machines):
   m.addConstr(sum(PT[i][j] * x[j] <= C[i] for j in range(num_products))
```

But this formulation contains an error. Repair the error, and modify it so that you can use it in your code.

8. You also need the sum function in your `setObjective` call. Complete your program until it runs. Then compare the outcome with your original product_mix.py. The results should match, otherwise there is a mistake somewhere.

3.2 Organize code in functions

The challenge here is to organize the code in a better way, that is, by means of functions. Functions are very useful for a number of reasons. The first is that function names (if properly chosen) act as documentation; a good function name describes what the code in the body does. The function also helps to hide complexity. A programmer just has to read the function name to guess what the function does; the programmer does not necessarily have to read all the code of the function. Finally, a function enables reuse; it is not necessary to copy the code time and again for nearly the same task but with different numbers: just call the function again with new arguments/parameters.

1. Make a new file product_mix_3.py.
2. Give this script the same functionality as the previous one, but now make a function `optimize` to which you pass the data arrays as argument. The output of the function should the optimal profit.
3. Run it to see that it works.

3.3 Inventory control

We will now implement the model of Section 16.2.1 of FP. As said, it is an interesting and practically useful case; students have been using it as part of their master thesis project to control the production and inventory levels at companies. You should realize that in industry problems, there can be like 1000 different product types, the holding costs and profits may depend on product type and time (e.g., discounts in summer time), the inventory positions run in the millions of items. Such cases are impossible to analyze with Excel; programming skills are absolutely necessary here[1].

1. Read FP Section 16.2.1 and the one paragraph intro of Section 16. We use the data of Section 16.2 to check our implementation.
2. Make a list with demand and capacities like so
```
D = [0, 80, 100, 120, 140, 90, 140]
C = [0, 100, 100, 100, 120, 120, 120]
```
   There is a reason for this that is slightly subtle. If you check FP, Eq 16.4, you'll see that to update the inventory level I_1 at time $t = 1$, you'll need the inventory level I_0 at time $t = 0$. Thus, in the implementation, the easiest is let all lists start at time 0 and put zeros where appropriate.
3. Add decision variables for X, I, and S. Note that they need to be 7 long, 6 for the number of periods t = 1,..., 6 , and 1 for period t = 0. Include in the definition of X, S and I that they should be non-negative, i.e., FP Eq. 16.5. Then include also that $X \leq C$ and $S \leq D$, i.e., Eqs 16.2 and 16.3.
4. Add constraints to ensure that $X[0] = 0$, $S[0] = 0$, and $I[0] = I_0$. In the specification of the problem in Section 16.2 $I_0 = 0$.

[1] In fact, as a general rule, the analysis of any realistic business problem cannot be carried out with Excel, or it is exceptionally hard (much, much harder than to write a python program and use libraries). So, once again, learn to program.

5. Finally add constraint 16.4 for every *t* with a for loop. Observe that we need 6 constraints in total. Observe also that if you do have to deal with weekly or daily demand data, for one year, than adding all constraints by typing is not particularly nice. You really need for loops here.
6. Finally, add objective Fp. Eq. 16.1; use `sum` or `quicksum` to make your life easy. Quicksum is a Gurobi specific version of sum, which is faster for optimization models. If you use it, make sure to also import the quicksum library.
7. Solve the problem, and compare your result with the output as shown in Figure 16.5.
8. Think a bit about how your excel sheet would look like if you would had to deal with yearly data. You should see that it would not fit on screen; in fact, it would look extremely messy.[2]
9. Write a function to read demand and capacity data from the file `inv_control_data_1.txt`. (First open this file with, e.g., Atom or notepad. It is best to check the format of an input file before you write code to read it.) Each line contains the demand and the capacity for a specific period. The data is the same as earlier, so testing must be easy. Note that if you read the file as csv and convert it to a list, the individual items in the list will be a string. You need to tell Python explicitly that the values are numbers, for example, `int(record[0])` if you want the values to be rounded to a whole number or `float(record[0])` if the numbers could contain fractions (e.g., 3.8).
10. Then read the info from `inv_control_data_2.txt`; this represents one year of weekly data. Solve the optimization problem. Interpret the results. When, for instance, does inventory start to build up? Are there weeks that we have lost sales? If so, is *r* sufficiently large? Can you modify the capacity planning to maximize the profit? (This is interesting challenge, and in fact, the challenge in many master thesis projects.)
11. An interesting optional extension is to try to make a graph of the inventory as a function of time. Graphs can be made with the `matplotlib` library.

Upload your inventory control model code to Nestor as end product of Week 2.

[2] As a general lesson again, Excel files are very hard to maintain and understand in comparison to (Python) programs. My code for this problem consists of some 15 lines, each of which is completely clear by itself.