Department of Mathematics and Computer Science
University of Southern Denmark, Odense

March 9, 2021
Marco Chiarandini

# DM545/DM871 – Linear and integer programming

## Lab, Spring 2021

---

## Introduction

The assignment aims at introducing the students to mathematical modeling languages. They allow to write in an easy and compact way problems in Linear Programming (LP), Integer Programming (IP) and Mixed Integer Linear Programming (MILP) and to feed them in proper forms to MILP solvers.
Well known mathematical programming languages are: AMPL, GAMS, ZIMPL and GNU MathProg. The last two are open source.
These languages are declarative type of languages, as opposed to procedural type. That is, we define the problem without saying how it should be solved. Moreover, they allow to separate the model from the data. In fact that is essentially all what they do, they *instantiate* the model on the given data. The output that is used by the solver can also be used for debugging purposes, that is, to check that the explicit model is as expected. There are two formats in which an instantiated MILP can be exported to a file: `.mps` format, which is an almost universal format among solver systems, but that is not easy to read and `.lp` format, which is more readable (introduced by CPLEX).
The primary solvers for MILP are:

- Gurobi (Version 9.1.1),
- FICO XPRESS (Version 8),
- IBM CPLEX (version 12.10).

These are commericial solvers. Commercial solvers remain maybe 6-7 time faster than the main free/open-source solvers:

- SCIP
- MIP-CL
- GLPK
- Coin-Cbc, part of the open-source initiative Coin-OR (`coin-or.org`)
- OR-Tools, Glop (only linear programming)

The NEOS server (`www.neos-server.org`) provides an infrastructure to upload an instantiated model and solve it remotedly.

In the course, we use Python with Gurobi. In Python we will import the module `gurobipy`, that makes available the library to pass models to the Gurobi solver. It is not a mathematical programming language but the library is very light and similar to a modelling language.
You can work on your own laptop, in which case it is enough to install the software there. If you work on the machines of the terminal room then install the software in your IMADA home directory.
This is a list of things to do before the class:

- install Python 3.6 (or later version): `http://python.org/download/`.
- install Gurobi:
  - download from `https://www.gurobi.com/downloads/`
  - register to get an academic license. It will last three months.
  - install Gurobi following the instructions at the Quick Start Guide relative to your system.

  - install the Gurobi Python Interface (`gurobipy`) following these instructions for mac, linux, windows (manual installation recommended)
  - run 'grbgetkey' using the argument provided at the license registration page (ex: grbgetkey ae36ac20-16e6-acd2-f242-4da6e765fa0a). The 'grbgetkey' program will prompt you to store the license file on your machine (to do this step you must be within the SDU Net via VPN connect).

- choose and prepare your favourite Python Integrated Development Environment (IDE): for example, Jupyter, JupyterLab, Spyder3, Atom (with the package hydrogen), Emacs, Eclipse, Visual Code, etc. If you are on Windows you may consider PyScripter IDE.

You can install all software listed above via the Anaconda Python distribution (a system-specific link is given from the Quick Start Guide linked above), which includes both a graphical development environment (Spyder) and a notebook-style interface (Jupyter). However, note that if you have already Python installed you may hand up having two distributions of it in your systems.

We assume that you have previous knowledge of Python programming (a couple of links to review Python programming are available from the course Web Page).
The reason for choosing Gurobi in this course is its excellent documentation:

- you can get an introduction to modeling in Python with gurobi watching at least the first 34 minutes of this video: Python I: Introduction to modeling with Python
- two other tutorials on LP and MILP are available
- an abundance of application cases
- examples
- the Application Program Interface (API).

In the remaining part of this document you will be guided through some elements of the course revisited with the use of gurobipy. Although the exercises are about implementing the models at the computer, remember that the best practice is to first write the mathematical models on paper! Do that for the subtasks that asks you to derive a mathematical model.

# Exercise 1 The Basics: Production Allocation

The following model is for a specific instance of the production allocation problem seen in the first lectures. We give here the primal and its dual model with the instantiated numerical parameters.

$$
\begin{array}{lrcrcrl}
\max & 5x_1 & + & 6x_2 & + & 8x_3 & = z \\
 & 6x_1 & + & 5x_2 & + & 10x_3 & \leq 60 \\
 & 8x_1 & + & 4x_2 & + & 4x_3 & \leq 40 \\
 & 4x_1 & + & 5x_2 & + & 6x_3 & \leq 50 \\
 & & & & & x_1, x_2, x_3 & \geq 0
\end{array}
\qquad
\begin{array}{lrcrcrl}
\min & 60y_1 & + & 40y_2 & + & 50y_3 & = u \\
 & 6y_1 & + & 8y_2 & + & 4y_3 & \leq 5 \\
 & 5y_1 & + & 4y_2 & + & 5y_3 & \leq 6 \\
 & 10y_1 & + & 4y_2 & + & 6y_3 & \leq 8 \\
 & & & & & y_1, y_2, y_3 & \geq 0
\end{array}
$$

## Analysis of the final tableau

As we will learn from theory solving one of the two problems will provide the solution also to the other problem. The primal solution is $x_1^* = 0, x_2^* = 7, x_3^* = 2.5$ and the dual solution is $y_1^* = 0.2, y_2^* = 0, y_3^* = 1$. The objective value is $z^* = u^* = 62$.
In the next introductory class we will learn to organize these values in a table called tableau. For the given problems it looks like this:

```
|------+----+----+------+----+----+----+-----|
|   x1 | x2 | x3 |   s1 | s2 | s3 | -z |   b |
|------+----+----+------+----+----+----+-----|
|   ?  |  1 |  0 |  ?   |  0 |  ?  |  0 |   7 |
|   ?  |  0 |  1 |  ?   |  0 |  ?  |  0 | 5/2 |
|   ?  |  0 |  0 |  ?   |  1 |  ?  |  0 |   2 |
```

```
|------+----+----+------+----+----+----+-----|
| -0.2 |  0 |  0 | -0.2 |  0 | -1 |  1 | -62 |
|------+----+----+------+----+----+----+-----|
```

A question mark is for the values that are not relevant for the goal of this exercise.
The three numbers of the last row in the tableau above in the columns of the variables that are not in basis are called *reduced costs*. They indicate how much we should make each product more expensive in order to be worth manufacturing it. The next three values are known as *shadow prices*. Aftr a change of sign they give the value of the dual variables, which are interpreted as the *marginal value* of increasing (or decreasing) the capacities of the resources (that is, the value by which the objective function would improve if the constraint were relaxed by one unit, which corresponds to buying one unit more of resource). In our example, which seeks maximization, the marginal value 1 for the third resource means that the objective function would increase by 1 if we could have one more unit of that resource. It can be verified that in the primal problem at the optimum the first and third resources are fully exhausted, that is, their constraint is satisfied at the equality, while there is *slack* for the second resource, that is, the constraint holds with strict inequality. Looking at the marginal values, we see that the second resource has been given a zero valuation. This seems plausible, since we are not using all the capacity that we have, we are not willing to place much value on it (buying one more unit of that resource would not translate in an improvement of the objective function).
These results are captured by the Complementary Slackness theorem of linear programming. If a constraint is not *"binding"* in the optimal primal solution, the corresponding dual variable is zero in the optimal solution to the dual model. Similarly, if a constraint in the dual model is not *"binding"* in the optimal solution to the dual model, then the corresponding variable is zero in the optimal solution to the primal model.

## Solving the model with Gurobi Python

Let's write the primal model in Python and solve it with Gurobi. Here is the script:

```
#!/usr/bin/python

from gurobipy import *

# Model
model = Model("prod")
model.setParam(GRB.param.Method, 0)

# Create decision variables
x1 = model.addVar(lb=0.0, ub=GRB.INFINITY, obj=5.0,
                  vtype=GRB.CONTINUOUS, name="x1") # arguments by name
x2 = model.addVar(0.0, GRB.INFINITY, 6.0, GRB.CONTINUOUS, "x2") # arguments by position
x3 = model.addVar(name="x3") # arguments by deafult

# The objective is to maximize (this is redundant now, but it will overwrite Var
    declaration
model.setObjective(5.0*x1 + 6.0*x2 + 8.0*x3, GRB.MAXIMIZE)

# Add constraints to the model
model.addConstr(6.0*x1 + 5.0*x2 + 10.0*x3 <= 60.0, "c1")
model.addConstr(8.0*x1 + 4.0*x2 + 4.0*x3, GRB.LESS_EQUAL, 40.0, "c2")
model.addConstr(4.0*x1 + 5.0*x2 + 6.0*x3, GRB.LESS_EQUAL, 50.0, "c3")

# Solve
model.optimize()

# Let's print the solution
for v in model.getVars():
    print(v.varName, v.x)
```

The documentation for the functions `Model.addVar()` and `Model.addConstr()`, as well as for all other functions in `gurobipy` is available from the Reference Manual and more specifically from the Model API

page. For the variable $x_3$ the lower bound, upper bound, objective coefficient and type are set to their default values that are, respectively: `lb=0.0, ub=GRB.INFINITY, obj=0.0, vtype=GRB.CONTINUOUS`. Once the model has been built, the typical next step is to optimize it (using `model.optimize`). You can then query the `x` attribute on the variables to retrieve the solution (and the `VarName` attribute to retrieve the variable name for each variable).
If the code is in a file called `prod1.py` then we can solve the model by calling:

```
> python prod1.py
```

If something goes wrong check that `gurobipy` is available at the import, that you have the license and that it is saved in the correct place and that there are no syntax errors. If everything runs fine you should get the following output:

```
Optimize a model with 3 rows, 3 columns and 9 nonzeros
Coefficient statistics:
  Matrix range     [4e+00, 1e+01]
  Objective range  [5e+00, 8e+00]
  Bounds range     [0e+00, 0e+00]
  RHS range        [4e+01, 6e+01]
Presolve time: 0.02s
Presolved: 3 rows, 3 columns, 9 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    1.9000000e+31   5.187500e+30   1.900000e+01      0s
       3    6.2000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 3 iterations and 0.04 seconds
Optimal objective  6.200000000e+01
x1 0.0
x2 7.0
x3 2.5
```

The screen output of Gurobi is described in the manual: console logging (simplex logging and MIP logging will be relevant for us). Gurobi reports first some statistics to check whether there is a need for rescaling conveniently the numbers of the problem (no need in this case) and then applies preprocessing techniques to assess whether the model can be trivially solved without running the simplex or whether it can be reduced. In our case none of the two are possible and we are left with 3 variables. Finally, the simplex method is applied and after 3 iterations of the primal simplex method (we set to use this method via `model.setParam(GRB.param.Method, 0)`) the optimal solution is found with value 62.

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the x variable attribute to be populated.
Attributes can be accessed in two ways in the Python interface. The first is to use the `getAttr()` and `setAttr()` methods, which are available on variables (`Var.getAttr`/`Var.setAttr`), linear constraints (`Constr.getAttr`/`Constr.setAttr`), and models (`Model.getAttr`/`Model.setAttr`). These are called with the attribute name as the first argument (e.g., `var.getAttr("x")` or `constr.setAttr("rhs", 0.0)`). Attributes can also be accessed more directly: you can follow an object name by a period, followed by the name of an attribute of that object. Note that upper/lower case is ignored when referring to attributes. Thus, `b = constr.rhs` is equivalent to `b = constr.getAttr("rhs")`, and `constr.rhs = 0.0` is equivalent to `constr.setAttr("rhs", 0.0)`. The full list of available attributes can be found following the link Attributes.

**The Value of the Dual variables** The value of the dual variables can be accessed by referring to the attribute `pi` of the corresponding constraints. In Python:

```
for c in model.getConstrs():
    print(c.constrName, c.pi)
```

to obtain

```
c1 0.2
c2 0.0
c3 1.0
```

These are the shadow prices (here 0.2, 0.0 and 1) which correspond to the marginal value of the resources. The c1 and c3 constraints' value is different from zero. This indicates that there's a variable on the upper bound for those constraints, or in other terms that these constraints are *"binding"*. The second constraint is not *"binding"*.

**Your Task** Try relaxing the value of each binding constraint one at a time, solve the modified problem, and see what happens to the optimal value of the objective function. Also check that, as expected, changing the value of non-binding constraints won't make any difference to the solution.

**Your Task** We can also access several quantities associated with the variables. A particularly relevant one is the *reduced cost*. Print the reduced costs of the variables for our example and make sure that they correspond to the expected values from the tableau above. [Hint: look for the variable attribute `rc`.] What can we say about the solution found on the basis of the reduced costs?

Let's now focus on the values output during the execution of the simplex. Let's first solve another small numerical example:

```
#!/usr/bin/python
from gurobipy import *

m = Model("infeas")

x = m.addVar(name="x") # ie, >= 0
y = m.addVar(name="y") # ie, >= 0

m.setObjective(x - y, GRB.MAXIMIZE)

m.addConstr(x + y <= 2, "c1")
m.addConstr(2*x + 2*y >= 5, "c2")

m.optimize()
```

Solving it we obtain:

```
Presolve removed 0 rows and 1 columns
Presolve time: 0.00s

Solved in 0 iterations and 0.00 seconds
Infeasible or unbounded model
```

This means that the presolve process has removed one column and identified the model as infeasible or unbounded. To discover which of the two is the case we need to run the simplex. To do so we remove the presolve process and resolve. We also ensure that we are using the primal simplex that makes things easier to interpret for us.
Gurobi allows to control these choice by means of model parameters. They include time limits or other execution halting controls. See Parameters for a reference list. The parameter `Presolve` controls whether presolving is used. The parameter `Method` controls which solution method is used. The default is the dual simplex. The other possibilities are 0=primal simplex, 1=dual simplex, 2=barrier, 3=concurrent, 4=deterministic concurrent. Let's add the following lines to our second small model.

```
if m.status == GRB.status.INF_OR_UNBD:
    # Turn presolve off to determine whether m is infeasible or unbounded
    m.setParam(GRB.param.Presolve, 0)
    m.setParam(GRB.param.Method, 0)
    m.optimize()
```

```
if m.status == GRB.status.OPTIMAL:
    print('Optimal objective: %g' % m.objVal)
    print( m.getVarByName("x").x )
    print( m.getVarByName("y").x )
    exit(0)
elif m.status != GRB.status.INFEASIBLE:
    print('Optimization was stopped with status %d' % m.status)
    exit(0)
```

Resolving gives us the needed information. The `Primal infeas.` is the objective function of the auxiliary problem in the phase I of the two-phase simplex method. The Primal Infeas. never reaches 0. This means that a feasible solution cannot be found and the problem is therefore infeasible. Try to change 5 to 2 in the right-hand-side of the second constraint of the model above. What happens? Explain the behaviour.

**Your Task** Resolve the production allocation example above using the primal method. Compare the iteration values of the simplex with the previous ones. Give your interpretation of the values Objective, Primal Inf. and Dual Inf.

**Exporting Model Data to a File** The function `model.write()` writes model data to a file. The file type is encoded in the suffix of the file name passed to the function. Valid suffixes for writing the model itself are `.mps`, `.rew`, `.lp`, or `.rlp`. The suffix for writing a solution is `.sol` while `.prm` for a parameter file.

**Your Task** Try all these formats on the production allocation example above and check their contents. The MPS file is not very user friendly. This is because the format is an old format when computer technology had much more limitations than nowadays. The CPLEX-LP format is a more explicit version of the problem that may be useful to check whether the model we implemented in Python is actually the one we intended.
If you have any of them installed, try solving the problem with other solvers, eg, `cplex`, `glpk` and `scip`. For this you have to use the MPS (Mathematical Programming System) format and run the following: [1]

```
cplex -c read prod.mps optimize
glpsol --mps prod.mps
scip -f prod.mps
```

Gurobi has also a command-line tool to solve model files:

```
gurobi_cl model.mps
```

You may also use the online solver at NEOS, the Network Enabled Optimization Server supported by the US federal government and located at Argonne National Lab. To submit an MPS model to NEOS visit http://www.neos-server.org/neos/, click on the icon "NEOS Solvers", scroll down to the Linear Programming or Mixed Integer Linear Programming list, click on one of those, scroll down to "Model File", click on "Choose File", select a file from your computer that contains an MPS model, scroll down to "e-mail address:", type in your email address, and click Submit to NEOS.

## Exercise 2 The Diet Example

So far we have written models with embedded data. However, when building an optimization model, it is typical to separate the optimization model itself from the data used to create an instance of the model. These two model ingredients are often stored in completely different files.

[1] Standard MPS files do not indicate whether to minimize or maximize the objective. Thus your MPS files will come out the same whether the objective is minimize or maximize.
As you are seeing, most solvers minimize the objective by default. A solver may have a switch to tell it to maximize instead, but that is different for each solver.
If you change the signs of all the objective coefficients, while leaving the constraints unchanged, then minimizing the resulting MPS file will be equivalent to maximizing the original problem. This can be done easily by putting the entire objective expression in parentheses and placing a minus sign in front of it.

There are alternate approaches to providing data to the optimization model: they can be embedded in the source file, read from an SQL database (using the Python sqlite3 package), or read them from an Excel spreadsheet (using the Python xlrd package) and more.

**Diet Problem** Bob wants to plan a nutritious diet, but he is on a limited budget, so he wants to spend as little money as possible. His nutritional requirements are as follows:
2000 Kcal
55 g protein
800 mg calcium
Bob is considering the following foods with corresponding nutritional values

| | Serving Size | Price per serving | Energy (Kcal) | Protein (g) | Calcium (mg) |
|---|---|---|---|---|---|
| Oatmeal | 28 g | 0.3 | 110 | 4 | 2 |
| Chicken | 100 g | 2.4 | 205 | 32 | 12 |
| Eggs | 2 large | 1.3 | 160 | 13 | 54 |
| Milk | 237 cc | 0.9 | 160 | 8 | 285 |
| Apple Pie | 170 g | 2 | 420 | 4 | 22 |
| Pork | 260 g | 1.9 | 260 | 14 | 80 |

With the help of Gurobi Python, find the amount of servings of each type of food in the diet.

In a file `dietmodel.py` we specify the model independently from the specific data. The file is reported in Listing 1. Make sure you understand the model, in particular, the global function `quicksum`.
We set the data in another file, for example, `diet1.py`, reported in Listing 2. The function `multidict` is another global function of Gurobi. Here is an example of what it does:

```
keys, dict1, dict2 = multidict( {
    'key1': [1, 2],
    'key2': [1, 3],
    'key3': [1, 4] } )
print keys, dict1, dict2
```

```
['key3', 'key2', 'key1']
{'key3': 1, 'key2': 1, 'key1': 1}
{'key3': 4, 'key2': 3, 'key1': 2}
```

The key construct that enables the separation of the model from the data is the Python module. A module is simply a set of functions and variables, stored in a file. One imports a module into a program using the import statement. One then executes the solve function of the dietmodel module by adding the following pair of statements at the end of the file `diet1.py`:

```
import dietmodel
dietmodel.solve(categories,minNutrition,maxNutrition,foods,cost,nutritionValues)
```

To solve the model we execute `diet1.py`.

**Your Task** A pill salesman offers Bob Calories, Protein, and Calcium pills to fulfill his nutritional needs. He needs to estimate the prices of units of serving, that is, the cost of 1 kcal, the cost of 1 g of protein, the cost of 1 mg of calcium. He wants to make as much money as possible, given Bob's constraints. He knows that Bob wants 2200 kcal, 55 g protein, and 1779 mg calcium. How can we help him in guaranteeing that he does not make a bad deal?

## Exercise 3 Particular Cases

The two following LP problems lead to two particular cases when solved by the simplex algorithm. Identify these cases and characterize them, that is, give indication of which conditions generate them in general. Then, implement the models in Gurobi Python and observe the behaviour.

Listing 1: `dietmodel.py`

```
#!/usr/bin/python
from pyscipopt import *


def solve(categories, minNutrition, maxNutrition, foods, cost, nutritionValues):
    # Model
    m = Model("diet")

    # Create decision variables for the nutrition information,
    # which we limit via bounds
    nutrition = {}
    for c in categories:
        nutrition[c] = m.addVar(lb=minNutrition[c], ub=maxNutrition[c], name=c)

    # Create decision variables for the foods to buy
    buy = {}
    for f in foods:
        buy[f] = m.addVar(obj=cost[f], name=f)

    # The objective is to minimize the costs
    m.setMinimize()

    # Nutrition constraints
    for c in categories:
        m.addCons(
            quicksum(nutritionValues[f, c] * buy[f] for f in foods) == nutrition[c],
            c)

    def printSolution():
        if m.getStatus() == "optimal":
            print('\nCost: %g' % m.getObjVal())
            print('\nBuy:')
            for f in foods:
                if m.getVal(buy[f]) > 0.0001:
                    print('%s %g' % (f, m.getVal(buy[f])))
            print('\nNutrition:')
            for c in categories:
                print('%s %g' % (c, m.getVal(nutrition[c])))
        else:
            print('No solution')

    # Solve
    m.writeProblem("diet.lp")
    m.optimize()
    printSolution()
```

Listing 2: `diet1.py`

```
#!/usr/bin/python

from pyscipopt import *

categories, minNutrition, maxNutrition = multidict({
  'Calories': [1800, 2200],
  'Protein': [91, None],
  'Calcium': [0, 1779] })

foods, cost = multidict({
  'Oatmeal':  0.30,
  'Chicken':  2.40,
  'Eggs':  1.30,
  'Milk':  0.90,
  'Apple Pie':  2.00,
  'Pork':  1.90});

# Nutrition values for the foods
nutritionValues = {
('Oatmeal', 'Calories' ): 110,
('Oatmeal', 'Protein' ): 4,
('Oatmeal', 'Calcium' ): 2,
('Chicken', 'Calories' ): 205,
('Chicken', 'Protein' ): 32,
('Chicken', 'Calcium' ): 12,
('Eggs', 'Calories' ): 160,
('Eggs', 'Protein' ): 13,
('Eggs', 'Calcium' ): 54,
('Milk', 'Calories' ): 160,
('Milk', 'Protein' ): 8,
('Milk', 'Calcium' ): 285,
('Apple Pie', 'Calories' ): 420,
('Apple Pie', 'Protein' ): 4,
('Apple Pie', 'Calcium' ): 22,
('Pork', 'Calories' ): 260,
('Pork', 'Protein' ): 14,
('Pork', 'Calcium' ): 80 };
```

$$
\begin{array}{rrcrcl}
\text{maximize} & 2x_1 & + & x_2 & & \\
\text{subject to} & & & x_2 & \leq & 5 \\
& -x_1 & + & x_2 & \leq & 1 \\
& & & x_1, x_2 & \geq & 0
\end{array}
$$

$$
\begin{array}{rrcrcl}
\text{maximize} & x_1 & + & x_2 & & \\
\text{subject to} & 5x_1 & + & 10x_2 & \leq & 60 \\
& 4x_1 & + & 4x_2 & \leq & 40 \\
& & & x_1, x_2 & \geq & 0
\end{array}
$$

## Exercise 4 Pathological Cases

This exercise asks you to check the behavior of the solvers on the two pathological cases:

$$
\begin{array}{rrcrcl}
\text{maximize} & & & 4x_2 & & \\
\text{subject to} & & & 2x_2 & \geq & 0 \\
& -3x_1 & + & 4x_2 & \geq & 1 \\
& & & x_1, x_2 & \geq & 0
\end{array}
$$

$$
\begin{array}{rrl}
\text{maximize} & 10x_1 - 57x_2 - 9x_3 - 24x_4 & \\
\text{subject to} & -0.5x_1 + 5.5x_2 + 2.5x_3 - 9x_4 & \leq 0 \\
& -0.5x_1 + 1.5x_2 + 0.5x_3 - x_4 & \leq 0 \\
& x_1 & \leq 1 \\
& x_1, x_2, x_3, x_4 & \geq 0
\end{array}
$$

What happens with the solver? Can you detect which pathological cases are from the output of the solver? How?

## Exercise 5 Shortest Path

Model the shortest path problem as an LP problem. Write the model in Python using the skeleton sp.py available from the directory http://www.imada.sdu.dk/~marco/DM559/Files/SP/20points.txt that reads data from 20points.tex. In these data the source is node 1 and the target is node 20.
Model the problem in LP and solve it with Gurobi Python. Check the correctness of your solution with the help of the visualization in the template `sp.py`.
It may be worth looking at the examples `netflow` and `tsp` from the Gurobi documentation to get inspiration for the model. For the implementation it may be helpful using the Gurobi list class tuplelist.