Gurobi tutorials for JBM035

Pieter Kleer

Table of contents

Installation

In this online course document we will use Gurobi to solve linear optimization problems via Python. In this chapter we outline the steps to install Gurobi.

It is assumed that you already have Python installed on your system. Otherwise, you need to install Python first. The Anaconda distribution is a good choice. If you get stuck anywhere in the installation process, then you can find more information on Gurobi's Quick Start Guide.

Register at Gurobi

Visit the Gurobi website and click on the register button. Open the registration form and make sure that you select the *Academic* account type, and select Student for the academic position.

Download Gurobi

Download Gurobi from the website. Note that you need to login with your Gurobi account before you can download Gurobi. Select the distribution that corresponds to your system (e.g., Windows or macOS), and select the regular "Gurobi Optimizer", not any of the AMPL variations. Unless mentioned otherwise, download the most recent version.

Install Gurobi

Run the installer and follow the installation steps. At some point, the installer may ask whether to add Gurobi to your execution path. This is probably useful to accept.

Gurobi license

You cannot use Gurobi without a license, so you need to apply for a license. As a student you can request a free academic license; take the **Named-User Academic** one. After you have obtained the license, you need to activate it for your Gurobi installation. If you open the license details on the Gurobi website, you can see what you need to do: open a command prompt and run the `grbgetkey` command with the code that corresponds to your license. This command will create your license file. Make sure that you remember where the license file is saved. The default location is probably the best choice.

Note that the `grbgetkey` command will check that you are on an academic domain, so you need to perform this step on the **university network** (possibly via a VPN connection). Once installed correctly, you can also run Gurobi without an active VPN connection.

Install the Python API

Gurobi should now be correctly installed, but we also want to be able to use it from Python. Therefore, we need to install Gurobi's Python package. Open your Anaconda prompt (if you have the Anaconda installation) and run the following commands.

- `conda config --add channels http://conda.anaconda.org/gurobi`
- `conda install gurobi`

If you don't have the Anaconda installation, then you can do something similar with the `pip` command.

Test your installation

Now you can test whether everything is setup correctly. Open an interactive Python session, for instance using Jupyter Notebook. Try the following commands:

```
from gurobipy import Model
model = Model()
```

If both these commands succeed, then you are done.

If the first command fails, then the Gurobi python module has not been installed correctly. If the second command fails, then the license has not been setup correctly (make sure the license file is at the right location).

Chapter 1

Basics

In this notebook, we will introduce the Gurobi solver and its Python API. It is assumed that you have already installed Gurobi on your system.

If at any point, you need more information, then can also go to the official Gurobi documentation online. Note that the documentation is also included in your local Gurobi installation. E.g., `C:/Program%20Files/gurobi810/win64/docs/refman/py_python_api_overview.html#sec:Python`

1.1 Gurobi module

If you look at the examples in the Gurobi documentation, then you will notice that the Gurobi is loaded in the global namespace:

```
from gurobipy import *
```

However, I would **not recommend** this, but instead make exactly clear what objects and functions we are using from the Gurobi module. Usually we need only a few objects and/or functions. The object that we will always need is the `Model`, so let's import that. Additionally, we will import `GRB`, which contains a collection of constants that we occassionally need.

```
from gurobipy import Model, GRB
```

1.2 Example: Duplo problem

Recall the Duplo problem from the lectures:

$$\begin{array}{lll} \text{maximize} & 15x_1 + 20x_2 & \text{(profit)} \\ \text{subject to} & x_1 + 2x_2 \leq 6 & \text{(big bricks)} \\ & 2x_1 + 2x_2 \leq 8 & \text{(small bricks)} \\ & x_1, x_2 \geq 0 & \end{array}$$

with

- x_1: number of chairs

- x_2: number of tables

Now let's implement the Duplo problem in Gurobi.

```
# Initialize Gurobi model.
model = Model('Duplo problem')
```

Next, we need to declare the variables. We can do this with the `addVar` method, which creates a `Var` object. Below you can see the `addVar` documentation. By default Gurobi assumes that a variable is continuous and non-negative, so we don't have to specify the `lb` (lower bound), `ub` (upper bound), and `vtype` arguments. It it usefull to add a `name` as we will see later.

```
help(Model.addVar)
```

```
ROUTINE:
  addVar(lb, ub, obj, vtype, name, column)

PURPOSE:
  Add a variable to the model.

ARGUMENTS:
  lb (float): Lower bound (default is zero)
  ub (float): Upper bound (default is infinite)
  obj (float): Objective coefficient (default is zero)
  vtype (string): Variable type (default is GRB.CONTINUOUS)
  name (string): Variable name (default is no name)
  column (Column): Initial coefficients for column (default is None)

RETURN VALUE:
  The created Var object.

EXAMPLE:
  v = model.addVar(ub=2.0, name="NewVar")
```

```
# Declare the two decision variables.
x1 = model.addVar(name='chairs')
x2 = model.addVar(name='tables')
```

Note that we also didn't use the `obj` argument to specify the objective coefficients. We use the `setObjective` method to do so by refering to the variables created above. The `sense` argument must be used to specify whether we want to maximize or minimize the objective function. For this, we use `GRB.MAXIMIZE` or `GRB.MINIMIZE`, respectively.

```
# Specifiy the objective function.
model.setObjective(15*x1 + 20*x2, sense=GRB.MAXIMIZE)
```

The next step is to declare the constraints using the `addConstr` method. Again we can specify the constraint using the variables, and we give the constraint a name as well.

```
# Add the resource constraints on bricks.
c1 = model.addConstr(x1 + 2*x2 <= 6, name='big-bricks')
c2 = model.addConstr(2*x1 + 2*x2 <= 8, name='small-bricks')
```

1.2.1 Export model to text file

Optionally, you can save your model in various formats, which allow you to inspect your model or export it to other solvers and/or modeling languages. One of these formats is the LP format. In the output, you can see the names we have used to declare the variables and constraints of the model.

```
# Save model as text file.
model.write('duplo.lp')
```

```
# Show file contents.
print(open('duplo.lp').read())
```

```
\ Model Duplo problem
\ LP format - for model browsing. Use MPS format to capture full model detail.
Maximize
  15 chairs + 20 tables
Subject To
 big-bricks: chairs + 2 tables <= 6
 small-bricks: 2 chairs + 2 tables <= 8
Bounds
End
```

1.2.2 Optimizing the model

Now the model is completely specified, we are ready to compute the optimal solution.

```
model.optimize()
```

```
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 2 columns and 4 nonzeros
Model fingerprint: 0xadc88607
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
  Objective range  [2e+01, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 8e+00]
```

```
Presolve time: 0.01s
Presolved: 2 rows, 2 columns, 4 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    3.5000000e+31   3.500000e+30   3.500000e+01      0s
       2    7.0000000e+01   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.02 seconds (0.00 work units)
Optimal objective  7.000000000e+01
```

We can see some output from Gurobi, and the last line tells us that Gurobi found an optimal solution with objective value 70. We can also access the objective value using the `ObjVal` attribute.

```
print('Objective value:', model.ObjVal)
```

```
Objective value: 70.0
```

Now, what is the optimal solution? We can obtain it by accessing the `X` *attribute* from the variables.

```
print('x1 = ', x1.X)
print('x2 = ', x2.X)
```

```
x1 =  2.0
x2 =  2.0
```

If the model has many variables, this is a cumbersome approach, and it's easier to iterate over all variables of the model using the `getVars` method.

```
for var in model.getVars():
    print(f'{var.VarName} = {var.X}')
```

```
chairs = 2.0
tables = 2.0
```

Exercise 1

Implement the Médecins sans Frontières (MSF) example, about building medical kits, from the slides of Lecture 1-2:

$$
\begin{array}{llrcrcrl}
\max & z = & x_1 & + & x_2 & & & \textit{total number of kits} \\
\text{s.t.} & & 250x_1 & + & 100x_2 & \leq & 3700 & \textit{budget constraint} \\
& & 5x_1 & + & 3x_2 & \leq & 80 & \textit{labor hours constraint} \\
& & & & x_2 & \leq & 15 & \textit{at most } 15 \textit{ vac. kits} \\
& & x_1, & & x_2 & \geq & 0 & \textit{nonneg. constraints}
\end{array}
$$

Exercise 2

Implement the Transportation problem example from the slides of Lecture 1-2:

$$
\begin{array}{lll}
\min & 131x_{11} + 405x_{12} + 188x_{13} + 396x_{14} + 485x_{15} + & \\
& 554x_{21} + 351x_{22} + 479x_{23} + 366x_{24} + 155x_{25} & \\
\text{s.t.} & x_{11} + x_{12} + x_{13} + x_{14} + x_{15} \leq 47 & \textit{supply Haarlem} \\
& x_{21} + x_{22} + x_{23} + x_{24} + x_{25} \leq 63 & \textit{supply Eindhoven} \\
& x_{11} + x_{21} \geq 28 & \textit{demand A'dam} \\
& x_{12} + x_{22} \geq 16 & \textit{demand Breda} \\
& x_{13} + x_{23} \geq 22 & \textit{demand Gouda} \\
& x_{14} + x_{24} \geq 31 & \textit{demand A'foort} \\
& x_{15} + x_{25} \geq 12 & \textit{demand Den Bosch} \\
& x_{11}, x_{12}, x_{13}, x_{14}, x_{15}, x_{21}, x_{22}, x_{23}, x_{24}, x_{25} \geq 0 & \textit{ship nonneg. amounts}
\end{array}
$$

1.3 Slack variables

An alternative is to collect the solution in a Python dictionary.

```
solution = {var.VarName: var.X for var in model.getVars()}
solution
```

```
{'chairs': 2.0, 'tables': 2.0}
```

What about the constraints? One thing we might want to check are the values of slack variables (Lecture 2). Note that behind the scenes, Gurobi has transformed the model to standard form by adding slack variables:

$$
\begin{array}{lll}
x_1 + 2x_2 + s_1 & = 6 & \text{(big bricks)} \\
2x_1 + 2x_2 + s_2 & = 8 & \text{(small bricks)}
\end{array}
$$

Below we can check that the slacks of both constraints are zero, from which we can conclude the constraints are binding (active), because the slacks are both zero.

```
c1.slack, c2.slack
```

```
(0.0, 0.0)
```

Instead of using our constraint references `c1` and `c2`, we can also iterate over all constraints using the `getConstrs` method:

```
{cons.ConstrName: cons.slack for cons in model.getConstrs()}
```

```
{'big-bricks': 0.0, 'small-bricks': 0.0}
```

1.4 Infeasible models

We already knew that the Duplo problem had an optimal solution, but sometimes a model can be infeasible or unbounded as well, as we saw in Lecture 3. The `status` attribute contains information about the outcome

of the solution process.

```
model.Status
```

```
2
```

The result is 2, which is quite unmeaningful. Below is a list of status codes. Hence, status code 2 means Gurobi has found an optimal solution. For more information, you can check their meaning online.

```
 1: LOADED
 2: OPTIMAL
 3: INFEASIBLE
 4: INF_OR_UNBD
 5: UNBOUNDED
 6: CUTOFF
 7: ITERATION_LIMIT
 8: NODE_LIMIT
 9: TIME_LIMIT
10: SOLUTION_LIMIT
11: INTERRUPTED
12: NUMERIC
13: SUBOPTIMAL
14: INPROGRESS
15: USER_OBJ_LIMIT
```

In your code, it might be useful to use a line such as:

```
if model.Status == GRB.OPTIMAL:
    print('We found an optimal solution.')
else:
    print('An optimal solution could not be found.')
    print('Status code:', model.Status)
```

```
We found an optimal solution.
```

Let's add a constraint that requires us to produce at least 10 chairs, which is impossible with the current resources.

```
c3 = model.addConstr(x1 >= 10, name='ten-chairs')

model.optimize()
```

```
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 3 rows, 2 columns and 5 nonzeros
Coefficient statistics:
  Matrix range     [1e+00, 2e+00]
```

```
  Objective range  [2e+01, 2e+01]
  Bounds range     [0e+00, 0e+00]
  RHS range        [6e+00, 1e+01]
Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    7.0000000e+01   8.000000e+00   0.000000e+00      0s

Solved in 1 iterations and 0.01 seconds (0.00 work units)
Infeasible model
```

```
  if model.Status == GRB.OPTIMAL:
      print('We found an optimal solution.')
  else:
      print('An optimal solution could not be found.')
      print('Status code:', model.Status)
```

```
An optimal solution could not be found.
Status code: 3
```

If we don't add a check about the model status, and just implement the solution, then weird things can happen. Below you can see that we can access the `X` attributes of the variables, but corresponding solution is *infeasible*.

```
  for var in model.getVars():
      print(f'{var.VarName} = {var.X}')
```

```
chairs = 10.0
tables = -6.0
```

Chapter 2

Beyond the basics

In the first notebook, we introduced the basics of Gurobi's Pyhton API using an **explicit** declaration of the Duplo problem. It is quite obvious that this approach is inconvenient for larger models, because it is inflexible and error-prone. For larger models, it makes sense to create a generic implementation that separates the model **data** and model **logic**.

Typically, in Python, scalar parameters will be stored in integer or float variables, and indexed parameters will be stored in dictionaries or lists. In this notebook, we will learn how to implement such a model in Python.

If at any point, you need more information, then can also go to the official Gurobi documention online.

2.1 Oil refinery problem

Recall the oil refinery problem from the exercises of *Lecture 1*[1]. Below we present its **model formulation**. Although in the Oil Refinery problem we only have three processes, two crudes and two products, we write the formulation down in a general form that can also be used in case there are more processes, crudes and/or products.

Sets and indices:

- Processes $(1, 2, 3)$: index j
- Crudes inputs (A, B): index i
- Products (gasoline, heating oil): index k

Parameters:

- c_j: cost of process j (euro/unit)
- a_{ij}: number of barrels input i needed per unit of process j
- b_i: number of barrels of input crude i available (in millions)
- r_{kj}: number of barrels output product k per unit of process j (in millions)
- p_k: sales price product k (euro/barrel)

Variables:

- x_j: number of times (in millions) proces j is used

[1] Exercise 1.16 from Bertsimas and Tsitsiklis (1997)

2.1.1 LP model

In this section we describe the objective function and constraints of the problem.

Objective: Our objective is the profit, which is the *sales* minus *production costs*:

- Sales: $\sum_k p_k \sum_j r_{kj} x_j$
- Costs: $\sum_j c_j x_j$

Constraints: We cannot use more input barrels than available:

$$\sum_j a_{ij} x_j \leq b_i \qquad \forall i,$$

and of course we have the non-negativity constraints.

Altogether, this leads to the following LOP:

$$\begin{aligned} \text{Maximize} \quad & \sum_k p_k \sum_j r_{kj} x_j - \sum_j c_j x_j & \\ \text{subject to} \quad & \sum_j a_{ij} x_j \leq b_i & \forall i = A, B \\ & x_j \geq 0 & \forall j = 1, 2, 3 \end{aligned}$$

2.1.2 Input data

All model parameters are related to the three sets processes, crudes, and products. We assume that this data is available as standard Python dictionaries. For larger models, data often has to be imported from data files in formats such as CSV, JSON, or Excel. Here, we will just define the data in Python.

```
#Cost of process i = 1,2,3
costs = {
    'process-1': 51,
    'process-2': 11,
    'process-3': 40,
}

#Amount of crude j = A,B needed for process i = 1,2,3
inputs = {
    ('crude A', 'process-1'): 3,
    ('crude A', 'process-2'): 1,
    ('crude A', 'process-3'): 5,
    ('crude B', 'process-1'): 5,
    ('crude B', 'process-2'): 1,
    ('crude B', 'process-3'): 3,
}

#Amount of product k = gas,oil coming out of process i = 1,2,3
outputs = {
    ('gasoline', 'process-1'): 4,
    ('gasoline', 'process-2'): 1,
```

```
    ('gasoline', 'process-3'): 3,
    ('heating oil', 'process-1'): 3,
    ('heating oil', 'process-2'): 1,
    ('heating oil', 'process-3'): 4,
}

#Available amount of crude j = A,B
resources = {
    'crude A': 8,
    'crude B': 5,
}

#Profit/sale price per product unit k = gas, oil
sales_price = {
    'gasoline': 38,
    'heating oil': 33,
}
```

The three sets (process, crudes, and products) can be obtained from the dictionary `keys`. This results in a `dict_keys` object.

```
# Obtain the sets from the dictionary keys.
processes = costs.keys()
crudes = resources.keys()
products = sales_price.keys()

print('Processes:', processes)
print('Crudes:', crudes)
print('Products:', products)
```

```
Processes: dict_keys(['process-1', 'process-2', 'process-3'])
Crudes: dict_keys(['crude A', 'crude B'])
Products: dict_keys(['gasoline', 'heating oil'])
```

One can convert a dict_keys object to, e.g., a list.

```
crudes_list = list(crudes)
print('First crude is', crudes_list[0], 'and second crude is', crudes_list[1], '.')
```

```
First crude is crude A and second crude is crude B .
```

2.2 Gurobi model

Now we are ready to declare the Gurobi model. First, we have to import some objects from the Gurobi module.

```
from gurobipy import Model, GRB, quicksum
```

```
model = Model('Oil refinery')
```

2.2.1 Decision variables

One possibility is to add the variables one by one, which we could do by iterating over the processes (i.e., process-1, process-2 and process-3).

```
x = {}
for j in processes:
    x[j] = model.addVar(name=j)
```

However, we can create all variables at once, using the `addVars` method of the `Model` object.

```
x = model.addVars(processes, name='process')
```

```
x
```

```
{'process-1': <gurobi.Var *Awaiting Model Update*>,
 'process-2': <gurobi.Var *Awaiting Model Update*>,
 'process-3': <gurobi.Var *Awaiting Model Update*>}
```

```
model.update()
```

```
x
```

```
{'process-1': <gurobi.Var process[process-1]>,
 'process-2': <gurobi.Var process[process-2]>,
 'process-3': <gurobi.Var process[process-3]>}
```

The variable `x` is a `tupledict`, which is a generalization of the standard Python dictionary. The `tupledict` offers some additional possibilities, but we will not discuss these in this course.

```
type(x)
```

```
gurobipy.tupledict
```

Linear expressions can be constructed using the standard `sum` function, however, this is rather inefficient if we are dealing with many variables. Therefore, it is preferred to use the Gurobi function `quicksum` for this purpose. For example, to get the sum of all the decision variables, we could use:

```
quicksum(x[j] for j in processes)
```

```
<gurobi.LinExpr: process[process-1] + process[process-2] + process[process-3]>
```

2.2.2 Objective function

We use the `quicksum` function to define the objective function.

```
model.setObjective(
    (
        quicksum(
            sales_price[k] * outputs[k, j] * x[j]
            for k in products for j in processes
        )
        - quicksum(costs[j] * x[j] for j in processes)
    ),
    sense=GRB.MAXIMIZE,
)
```

```
model.getObjective()
```

```
<gurobi.LinExpr: 0.0>
```

If we update the model, then we can see that Gurobi automatically computes the (simplified) coefficients for us. That is, the original objective written out is

$$38(4x_1 + x_2 + 3x_3) + 33(3x_1 + x_2 + 4x_3) - (51x_1 + 11x_2 + 40x_3)$$

but this is automatically simplified to $200x_1 + 60x_2 + 206x_3$.

```
model.update()
model.getObjective()
```

```
<gurobi.LinExpr: 200.0 process[process-1] + 60.0 process[process-2] + 206.0 process[process-
3]>
```

We can also add multiple constraints of the same type using the `addConstrs` method. Here we specify a constraints for every element in the set `crudes`.

```
c = model.addConstrs(
    (
        quicksum(inputs[i, j] * x[j] for j in processes) <= resources[i]
        for i in crudes
    ),
    name='capacity',
)
```

```
c
```

```
{'crude A': <gurobi.Constr *Awaiting Model Update*>,
 'crude B': <gurobi.Constr *Awaiting Model Update*>}
```

```
model.update()
```

```
c
```

```
{'crude A': <gurobi.Constr capacity[crude A]>,
 'crude B': <gurobi.Constr capacity[crude B]>}
```

We can review the constraint properties using the `getRow` method and the `RHS` and `sense` attributes as illustrated below.

```
model.getRow(c['crude A'])
```

```
<gurobi.LinExpr: 3.0 process[process-1] + process[process-2] + 5.0 process[process-
3]>
```

```
c['crude A'].RHS
```

```
8.0
```

```
c['crude A'].sense
```

```
'<'
```

Now the model is fully specified and we can solve it.

```
model.optimize()
```

```
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)

CPU model: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 6 columns and 6 nonzeros
Model fingerprint: 0x8b2f8c5b
Coefficient statistics:
  Matrix range     [1e+00, 5e+00]
  Objective range  [6e+01, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [5e+00, 8e+00]
Presolve removed 0 rows and 3 columns
Presolve time: 0.01s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    5.3333333e+02   2.706333e+00   0.000000e+00      0s
       2    3.3900000e+02   0.000000e+00   0.000000e+00      0s

Solved in 2 iterations and 0.02 seconds (0.00 work units)
```

```
Optimal objective  3.390000000e+02
```

```
model.ObjVal
```

```
339.0
```

2.3 Model in function

We can declare the Gurobi model in a function, which clearly illustrate the separation between the model logic and model data. This will allow us to easily solve similar problems with other processes and inputs/outputs.

```
def oil_refinery(costs, inputs, outputs, resources, sales_price):
    """Return Gurobi model for the Oil Refinery problem."""
    # Obtain the sets from the dictionary keys.
    processes = costs.keys()
    crudes = resources.keys()
    products = sales_price.keys()

    model = Model('Oil refinery')

    x = model.addVars(processes, name='process')

    model.setObjective(
        (
            quicksum(
                sales_price[k] * outputs[k, j] * x[j]
                for k in products for j in processes
            )
            - quicksum(costs[j] * x[j] for j in processes)
        ),
        sense=GRB.MAXIMIZE,
    )

    model.addConstrs(
        (
            quicksum(inputs[i, j] * x[j] for j in processes) <= resources[i]
            for i in crudes
        ),
        name='capacity',
    )
    return model
```

```
model = oil_refinery(costs, inputs, outputs, resources, sales_price)
model.optimize()
```

```
Gurobi Optimizer version 10.0.1 build v10.0.1rc0 (win64)
```

```
CPU model: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz, instruction set [SSE2|AVX|AVX2]
Thread count: 4 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 3 columns and 6 nonzeros
Model fingerprint: 0xb4151a6b
Coefficient statistics:
  Matrix range     [1e+00, 5e+00]
  Objective range  [6e+01, 2e+02]
  Bounds range     [0e+00, 0e+00]
  RHS range        [5e+00, 8e+00]
Presolve time: 0.01s
Presolved: 2 rows, 3 columns, 6 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0    8.8600000e+32   4.000000e+30   8.860000e+02      0s
       3    3.3900000e+02   0.000000e+00   0.000000e+00      0s

Solved in 3 iterations and 0.01 seconds (0.00 work units)
Optimal objective  3.390000000e+02
```

```python
print(f'Profit: {model.ObjVal}')
for var in model.getVars():
    print(f'{var.VarName} = {var.X}')
```

```
Profit: 339.0
process[process-1] = 0.0
process[process-2] = 0.5
process[process-3] = 1.5
```

2.4 Multi-dimensional indices

Using the same functionality as above, we can also create multi-dimensional decision variables. Suppose that we have a list of factories and a list of products:

```python
factories = ['factory-1', 'factory-2']
products = ['product-1', 'product-2', 'product-3']
```

Our decision variables x_{ij} denote how many units of product j should be produced in factory i. This decision variable has two indices: i and j. We can use the `addVars` method to declare these variables by specifying both lists as arguments. This automatically creates all factory-product combinations.

```python
model = Model()

x = model.addVars(factories, products, name='production')
model.update()
x
```

```
{('factory-1', 'product-1'): <gurobi.Var production[factory-1,product-1]>,
 ('factory-1', 'product-2'): <gurobi.Var production[factory-1,product-2]>,
 ('factory-1', 'product-3'): <gurobi.Var production[factory-1,product-3]>,
 ('factory-2', 'product-1'): <gurobi.Var production[factory-2,product-1]>,
 ('factory-2', 'product-2'): <gurobi.Var production[factory-2,product-2]>,
 ('factory-2', 'product-3'): <gurobi.Var production[factory-2,product-3]>}
```

You can access an individual variable by specifying the indices.

```
x['factory-1', 'product-2']
```

```
<gurobi.Var production[factory-1,product-2]>
```

Using `quicksum`, the multi-dimensional variable can be easily used to declare the objective or constraints.

```
sum_factory_1 = quicksum(x['factory-1', j] for j in products)
sum_factory_1
```

```
<gurobi.LinExpr: production[factory-1,product-1] + production[factory-1,product-
2] + production[factory-1,product-3]>
```

If you want to have more control over which combinations factories and products must be created, you can also create the list of combinations, and use that when declaring the variables.

Suppose that `factory-1` cannot produce `product-1`. Then we can do the following.

```
# List of all tuple combinations of factories and products.
combinations = [(i, j) for i in factories for j in products]
combinations
```

```
[('factory-1', 'product-1'),
 ('factory-1', 'product-2'),
 ('factory-1', 'product-3'),
 ('factory-2', 'product-1'),
 ('factory-2', 'product-2'),
 ('factory-2', 'product-3')]
```

```
# Remove non-applicable tuple.
combinations.remove(('factory-1', 'product-1'))
combinations
```

```
[('factory-1', 'product-2'),
 ('factory-1', 'product-3'),
 ('factory-2', 'product-1'),
 ('factory-2', 'product-2'),
 ('factory-2', 'product-3')]
```

```
# Use list of tuples to declare decision variables.
y = model.addVars(combinations, name='production')
```

```
model.update()
y
```

```
{('factory-1', 'product-2'): <gurobi.Var production[factory-1,product-2]>,
 ('factory-1', 'product-3'): <gurobi.Var production[factory-1,product-3]>,
 ('factory-2', 'product-1'): <gurobi.Var production[factory-2,product-1]>,
 ('factory-2', 'product-2'): <gurobi.Var production[factory-2,product-2]>,
 ('factory-2', 'product-3'): <gurobi.Var production[factory-2,product-3]>}
```

Exercise 3

Consider the general transportation problem for m plants and n customers from Lec1-2.

$$
\begin{array}{lll}
\text{min} & \sum_{p=1}^{m}\sum_{c=1}^{n} U_{pc}x_{pc} & \textit{total transportation costs} \\
\text{s.t.} & \sum_{c=1}^{n} x_{pc} \le S_p \quad \forall p & \textit{supply available at each p(lant) is } S_p \\
 & \sum_{p=1}^{m} x_{pc} \ge D_c \quad \forall c & \textit{demand } D_c \textit{ met for each c(ustomer)} \\
 & x_{pc} \ge 0 \quad \forall (p,c) & \textit{ship nonneg. amounts}
\end{array}
$$

Write a function that as input a general cost matrix `c` (list of lists), a list of supplies `S` for the plants, and a list of demands `D` for the customers. It should output the transportation problem model above. Test your function on the input from Exercise 2 of Tutorial 1, i.e.,

$$c = [[131, 405, 188, 396, 485], [554, 351, 479, 366, 155]], S = [47, 63],$$

and

$$D = [28, 16, 22, 31, 12].$$