gurobipy

Course

Xavier Nodet, xavier.nodet@gurobi.com

January 2025 - e0178e0

Latest update of the course, slides and exercises

https://nodet.github.io

Gurobi can solve models with linear and quadratic (incl. nonconvex) constraints and objective.

$$
\begin{array}{rll}
\text{minimize} & x^T Q x + c^T x + d & \\
\text{subject to} & Ax = b & \\
& x^T Q_i x + c_i^T x \leq d_i & \forall\ i \in I \\
& l \leq x \leq u & \\
& x_j \in \mathbb{Z} & \forall\ j \in J
\end{array}
$$

And nonlinear constraints as well!

1. Introductory examples

Let us start with a simple example

$$
\begin{array}{ll}
\text{maximize} & x + y + 2z \\
\text{subject to} & x + 2y + 3z \le 4 \\
& x + y \ge 1 \\
& x,\ y,\ z \in \{0, 1\}
\end{array}
$$

Here is the Python code to solve this problem:

```python
import gurobipy as gp                                                      ①
from gurobipy import GRB                                                   ②

with gp.Env() as env, gp.Model("simple-example", env=env) as model:       ③
    x = model.addVar(vtype=GRB.BINARY, name="x")                           ④
    y = model.addVar(vtype=GRB.BINARY, name="y")
    z = model.addVar(vtype=GRB.BINARY, name="z")

    model.addConstr(x + 2 * y + 3 * z <= 4, name="c0")                     ⑤
    model.addConstr(x + y >= 1, name="c1")

    model.setObjective(x + y + 2 * z, sense=GRB.MAXIMIZE)                  ⑥

    model.write("example.lp")                                              ⑦
```

```python
    model.optimize()                                     ⑧

    print("******* Solution *******")
    for var in model.getVars():
        print(f"{var.VarName}: {var.X}")                 ⑨
    print("************************")
```

① Import gurobipy package as gp for convenience

② GRB is the list of all Gurobi constants

③ Create a Gurobi environment and a model object

④ Define decision variables

⑤ Define constraints

⑥ Define objective

⑦ Save the model as an LP file

⑧ Optimize model

⑨ X attribute is the variable's value in the solution

Here is the log of the execution of this program:

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (mac64[arm] - Darwin 24.2.0
24C101)

CPU model: Apple M1 Pro
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0x98886187
Variable types: 0 continuous, 3 integer (3 binary)
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [1e+00, 2e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 4e+00]
```

```
Found heuristic solution: objective 2.0000000
Presolve removed 2 rows and 3 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds (0.00 work units)
Thread count was 1 (of 8 available processors)

Solution count 2: 3 2

Optimal solution found (tolerance 1.00e-04)
Best objective 3.000000000000e+00, best bound 3.000000000000e+00, gap 0.0000%
******* Solution *******
x: 1.0
y: 0.0
z: 1.0
```

```
************************
```

And here’s the generated LP file:

```
\ Model simple-example
\ LP format - for model browsing. Use MPS format to capture full model detail.
\ Signature: 0xd6af213f17f735ae
Maximize
  x + y + 2 z
Subject To
 c0: x + 2 y + 3 z <= 4
 c1: x + y >= 1
Bounds
Binaries
 x y z
End
```

The same example, using the matrix API

```
import gurobipy as gp
from gurobipy import GRB
import numpy as np
import scipy.sparse as sp

with gp.Env() as env, gp.Model("matrix1", env=env) as m:

    # Create variables
    x = m.addMVar(shape=3, vtype=GRB.BINARY, name="x")

    # Set objective
    obj = np.array([1.0, 1.0, 2.0])
    m.setObjective(obj @ x, GRB.MAXIMIZE)
```

```python
# Build (sparse) constraint matrix
row = np.array([0, 0, 0, 1, 1])
col = np.array([0, 1, 2, 0, 1])
val = np.array([1.0, 2.0, 3.0, -1.0, -1.0])
# A is such that A[row[k], col[k]] = val[k]
A = sp.csr_matrix((val, (row, col)), shape=(2, 3))

# Build rhs vector
rhs = np.array([4.0, -1.0])

# Add constraints
m.addConstr(A @ x <= rhs, name="c")

# Write the model
m.write("matrix1.lp")

# Optimize model
```

```
    m.optimize()

    print(x.X)
    print(f"Obj: {m.ObjVal:g}")
```

Here's the log of the execution:

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (mac64[arm] - Darwin 24.2.0
24C101)

CPU model: Apple M1 Pro
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 2 rows, 3 columns and 5 nonzeros
Model fingerprint: 0x8d4960d3
Variable types: 0 continuous, 3 integer (3 binary)
Coefficient statistics:
  Matrix range     [1e+00, 3e+00]
  Objective range  [1e+00, 2e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [1e+00, 4e+00]
```

```
Found heuristic solution: objective 2.0000000
Presolve removed 2 rows and 3 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds (0.00 work units)
Thread count was 1 (of 8 available processors)

Solution count 2: 3 2

Optimal solution found (tolerance 1.00e-04)
Best objective 3.000000000000e+00, best bound 3.000000000000e+00, gap 0.0000%
[1. 0. 1.]
Obj: 3
```

And here’s the generated LP file:

```
\ Model matrix1
\ LP format - for model browsing. Use MPS format to capture full model detail.
\ Signature: 0xd6af213f17f735ae
Maximize
  x[0] + x[1] + 2 x[2]
Subject To
 c[0]: x[0] + 2 x[1] + 3 x[2] <= 4
 c[1]: - x[0] - x[1] <= -1
Bounds
Binaries
 x[0] x[1] x[2]
End
```

2. Python Data Structures

- `tuple`: An ordered, compound grouping that cannot be modified once it is created and is ideal for representing multi dimensional subscripts.

  ```
  ("city_0", "city_1")
  ```

- `list`: An ordered group, so each item is indexed. Lists can be modified by adding, deleting or sorting elements.

  ```
  ["city_0", "city_1", "city_2"]
  ```

- `set`: An unordered group of unique elements. Sets can only be modified by adding or deleting.

```
{"city_0", "city_1", "city_2"}
```

- `dict`: A key-value pair mapping that is ideal for representing indexed data such as cost, demand, capacity.

```
demand = {"city_0": 100, "city_1": 50, "city_2": 40}
```

3. Extended Data Structures in gurobipy

3.1. tuplelist

`tuplelist`: a sub-class of Python list, used to build sub-lists efficiently. See in particular `tuplelist.select(pattern)`.

```
import gurobipy as gp
l = gp.tuplelist([('A', 'B', 'C'),
                  ('A', 'C', 'D'),
                  ('A', 'E', 'C')])
print(l.select('A', '*', 'C'))
```

```
<gurobi.tuplelist (2 tuples, 3 values each):
 ( A , B , C )
```

```
 ( A , E , C )
>
```

3.2. tupledict

`tupledict`: a sub-class of Python dict, where the values are usually Gurobi variables, to efficiently retrieve those whose key match a specified tuple pattern.

Some important methods to build linear expressions efficiently:

- tupledict.select(pattern) → `list`
- tupledict.sum(pattern) → `gp.LinExpr`
- tupledict.prod(coeff, pattern) → `gp.LinExpr`

```
import gurobipy as gp
m = gp.Model()
```

```python
x = m.addVars([(1,2), (1,3), (2,3)], name="x")     # x is a tupledict
m.update()                                          # Process all model updates
expr = x.sum('*', 3)
print(expr)
```

```
x[1,3] + x[2,3]
```

3.3. multidict()

multidict() is a convenience function to split a dict of lists.

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

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

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

3.4. Example of extended structures

```python
import gurobipy as gp
from gurobipy import GRB

data = gp.tupledict([
        (("a", "b", "c"), 3),
        (("a", "c", "b"), 4),
        (("b", "a", "c"), 5),
        (("b", "c", "a"), 6),
        (("c", "a", "b"), 7),
        (("c", "b", "a"), 3)
    ])
print(f"data: {data}")

print("\nTuplelist:")
```

```python
keys = gp.tuplelist(data.keys())
print(f"\tselect: {keys.select('a', '*', '*')}")

print("\nTupledict:")
print(f"\tselect  : {data.select('a', '*', '*')}")
print(f"\tsum     : {data.sum('*', '*', '*')}")
coeff = {("a", "c", "b"): 6, ("b", "c", "a"): -4}
print(f"\tprod    : {data.prod(coeff, '*', 'c', '*')}")

arcs, capacity, cost = gp.multidict({
        ("Detroit ", "Boston "): [100, 7],
        ("Detroit ", "New York "): [80, 5],
        ("Detroit ", "Seattle "): [120, 4],
        ("Denver ", "Boston "): [120, 8],
        ("Denver ", "New York "): [120, 11],
        ("Denver ", "Seattle "): [120, 4],
    })
```

```python
print("\nMultidict:")
print(f"\tcost: {cost}")
print("\n")
print(f"\tcapacity: {capacity}")
```

```
data: {('a', 'b', 'c'): 3, ('a', 'c', 'b'): 4, ('b', 'a', 'c'): 5, ('b', 'c',
'a'): 6, ('c', 'a', 'b'): 7, ('c', 'b', 'a'): 3}

Tuplelist:
    select: <gurobi.tuplelist (2 tuples, 3 values each):
 ( a , b , c )
 ( a , c , b )
>

Tupledict:
    select  : [3, 4]
```

```
    sum     : 28.0
    prod    : 0.0

Multidict:
    cost: {('Detroit ', 'Boston '): 7, ('Detroit ', 'New York '): 5, ('Detroit
', 'Seattle '): 4, ('Denver ', 'Boston '): 8, ('Denver ', 'New York '): 11,
('Denver ', 'Seattle '): 4}


    capacity: {('Detroit ', 'Boston '): 100, ('Detroit ', 'New York '): 80,
('Detroit ', 'Seattle '): 120, ('Denver ', 'Boston '): 120, ('Denver ', 'New
York '): 120, ('Denver ', 'Seattle '): 120}
```

4. Environments

Environments hold data that is global to one or more models.

- They hold a Gurobi license.
- They capture sets of parameter settings.
- They delineate a (single-threaded) Gurobi session.

The basic usage pattern is the following:

```python
import gurobipy as gp
from gurobipy import GRB

with gp.Env() as env, gp.Model("name", env=env) as m:
    # Use the model
    ...
```

A more advanced usage pattern is:

```python
import gurobipy as gp
from gurobipy import GRB

with gp.Env(empty=True) as env:
    # Set licensing parameters
    env.setParam("CloudAccessID", "...")
    env.setParam("CloudSecretKey", "...")
    env.setParam("LicenseID", ...)
    # Start the environment before creating a model
    env.start()

    with gp.Model("name", env=env) as m:
        # Use the model
        ...
```

5. Models

A model holds:

- variables
- constraints
- parameters, that define the behavior of the solver

```python
with gp.Env() as env, gp.Model("simple-example", env=env) as model:
    x = model.addVar(vtype=GRB.BINARY, name="x")
    y = model.addVar(vtype=GRB.BINARY, name="y")
    c1 = model.addConstr(x + y >= 1, name="c1")
    model.setObjective(x + y + 2 * z, sense=GRB.MAXIMIZE)

    model.params.MipFocus=1                                         ①
    model.params.TimeLimit = 3600                                   ②

    model.optimize()
```

① Focus on finding the best possible solutions

② Stop after one hour

It has methods to create and edit variables and constraints, to set parameters, to solve the model, to retrieve information, and more.

```
# ...
model.write("model.mps")             ①
model.chgCoeff(c1, x, 2)             ②
model.computeIIS()                   ③
```

① Store the model in an MPS file

② Change the coefficient of variable `x` in constraint `c1` to 2

③ Compute an Irreducible Inconsistent Set

5.1. Decision Variables, Model.addVar()

A decision variable is necessarily associated to exactly one instance of Model, and gets created using methods such as Model.addVar() to create a single variable, Model.addVars() to create multiple variables at once and Model.addMVars() to create a matrix of variables.

```
Model.addVar(lb=0.0, ub=float('inf'),
             obj=0.0,
             vtype=GRB.CONTINUOUS,
             name="")
```

The available variable types in Gurobi are:

- Continuous: GRB.CONTINUOUS
- General integer: GRB.INTEGER
- Binary: GRB.BINARY
- Semi-continuous: GRB.SEMICONT
- Semi-integer: GRB.SEMIINT

A semi-continuous variable has the property that it takes a value of 0, or a value between the specified lower and upper bounds. A semi-integer variable adds the additional restriction that the variable should take an integral value.

```python
# Define a binary decision variable with (default) lb=0
x = model.addVar(vtype=GRB.BINARY, name="x")
# Define an integer variable with lb=-1, ub=100
y = model.addVar(lb=-1, ub=100, vtype=GRB.INTEGER, name="y")
```

5.2. Model.addVars()

To add multiple decision variables to the model, use the Model.addVars() method which returns a Gurobi tupledict object containing the newly created variables:

```
Model.addVars(*indices,
              lb=0.0, ub=float('inf'),
              obj=0.0,
              vtype=GRB.CONTINUOUS,
              name="")
```

The first argument is an iterable giving indices for accessing the variables:

- several integers (specifying the dimensions of the matrix)
- several lists of scalars (each list specifies indices across one dimension of the matrix)
- one list of tuples, or a tuplelist

When the given name is a single string, it is subscripted by the index of the generator expression. The names are stored as ASCII strings, you should not use non-ASCII characters or spaces.

```python
import gurobipy as gp
from gurobipy import GRB

with gp.Model(name="model") as model:
    # 3D array of binary variables
    x = model.addVars(2, 3, 4, vtype=GRB.BINARY, name="x")
    model.update()
    print(model.getAttr("VarName", model.getVars()))
```

```
['x[0,0,0]', 'x[0,0,1]', 'x[0,0,2]', 'x[0,0,3]', 'x[0,1,0]', 'x[0,1,1]',
'x[0,1,2]', 'x[0,1,3]', 'x[0,2,0]', 'x[0,2,1]', 'x[0,2,2]', 'x[0,2,3]',
'x[1,0,0]', 'x[1,0,1]', 'x[1,0,2]', 'x[1,0,3]', 'x[1,1,0]', 'x[1,1,1]',
'x[1,1,2]', 'x[1,1,3]', 'x[1,2,0]', 'x[1,2,1]', 'x[1,2,2]', 'x[1,2,3]']
```

```python
import gurobipy as gp
from gurobipy import GRB

with gp.Model(name="model") as model:
    # Use arbitrary lists of immutable objects -> tupledict
    y = model.addVars([1, 5], [7, 3, 2], ub=range(6),
                      name=[f"y_{i}" for i in range(6)])
    model.update()
    print("\nVariables names, upper bounds, and indices:")
    for index, var in y.items():
        print(f"name: {var.VarName}, ub: {var.UB}, index: {index}")
```

```
Variables names, upper bounds, and indices:
name: y_0, ub: 0.0, index: (1, 7)
name: y_1, ub: 1.0, index: (1, 3)
name: y_2, ub: 2.0, index: (1, 2)
```

```
name: y_3, ub: 3.0, index: (5, 7)
name: y_4, ub: 4.0, index: (5, 3)
name: y_5, ub: 5.0, index: (5, 2)
```

```python
import gurobipy as gp
from gurobipy import GRB

with gp.Model(name="model") as model:
    # Use arbitrary list of tuples as indices
    z = model.addVars(
        [(3, "a"), (3, "b"), (7, "b"), (7, "c")], name="z",
    )
    model.update()
    print("\nVariables names and lower and upper bounds:")
    for index, var in z.items():
        print(f"name: {var.VarName}, lb: {var.LB}, ub: {var.UB}")
```

```
Variables names and lower and upper bounds:
name: z[3,a], lb: 0.0, ub: inf
name: z[3,b], lb: 0.0, ub: inf
```

```
name: z[7,b], lb: 0.0, ub: inf
name: z[7,c], lb: 0.0, ub: inf
```

5.3. Constraints, Model.addConstr()

Like variables, constraints are also associated with a model. Use the method Model.addConstr() to add a constraint to a model.

```
Model.addConstr(constr, name="")
```

`constr` is a TempConstr object that can take different types:

- Linear Constraint: `x + y <= 1`
- Ranged Linear Constraint: `x + y == [1, 3]`
- Quadratic Constraint: `x*x + y*y + x*y <= 1`
- Linear Matrix Constraint: `A @ x <= 1`
- Quadratic Matrix Constraint: `x @ Q @ y <= 2`
- Absolute Value Constraint: `x == abs_(y)`
- Logical Constraint: `x == and_(y, z)`
- Min or Max Constraint: `z == max_(x, y, constant=9)`
- Indicator Constraint: `(x == 1) >> (y + z <= 5)`

```python
import gurobipy as gp
from gurobipy import GRB

# Add constraint "\sum_{i=0}^{n-1} x_i <= b" for any given n and b.
n, b = 10, 4
with gp.Model("model") as model:
    x = model.addVars(n, vtype=GRB.BINARY, name="x")
    c1 = model.addConstr(x.sum() <= b, name="c1")
    model.update()

    print(f"RHS, sense = {c1.RHS}, {c1.Sense}")
    print(f"row: {model.getRow(c1)}")
```

```
RHS, sense = 4.0, <
row: x[0] + x[1] + x[2] + x[3] + x[4] + x[5] + x[6] + x[7] + x[8] + x[9]
```

```python
import gurobipy as gp
from gurobipy import GRB

# Add constraints "x_i + y_j - x_i * y_j >= 3".
n, m = 3, 2
with gp.Model("model") as model:
    x = model.addVars(n, name="x")
    y = model.addVars(m, name="y")
    for i in range(n):
        for j in range(m):
            model.addConstr(x[i] + y[j] - x[i] * y[j] >= 3, name=f"c_{i}{j}")
    model.update()

    for c in model.getQConstrs():
        print(f"Name: {c.QCName}, RHS: {c.QCRHS}, sense: {c.QCSense}")
        print(f"\trow: {model.getQCRow(c)}")
```

```
Name: c_00, RHS: 3.0, sense: >
    row: x[0] + y[0] + [ -1.0 x[0] * y[0] ]
Name: c_01, RHS: 3.0, sense: >
    row: x[0] + y[1] + [ -1.0 x[0] * y[1] ]
Name: c_10, RHS: 3.0, sense: >
    row: x[1] + y[0] + [ -1.0 x[1] * y[0] ]
Name: c_11, RHS: 3.0, sense: >
    row: x[1] + y[1] + [ -1.0 x[1] * y[1] ]
Name: c_20, RHS: 3.0, sense: >
    row: x[2] + y[0] + [ -1.0 x[2] * y[0] ]
Name: c_21, RHS: 3.0, sense: >
    row: x[2] + y[1] + [ -1.0 x[2] * y[1] ]
```

5.4. Model.addConstrs()

To add multiple constraints to the model, use the Model.addConstrs() method which returns a Gurobi tupledict that contains the newly created constraints:

```
Model.addConstrs(generator, name="")
```

```python
import gurobipy as gp
from gurobipy import GRB

I = range(2)
J = ["a", "b", "c"]
with gp.Model("model") as model:
    x = model.addVars(I, name="x")
    y = model.addVars(J, name="y")

    # Add constraints x_i + y_j <= 1 for all (i, j)
    model.addConstrs((x[i] + y[j] <= 1 for i in I for j in J), name="c")
    model.update()
    print(model.getAttr("ConstrName", model.getConstrs()))
```

```
['c[0,a]', 'c[0,b]', 'c[0,c]', 'c[1,a]', 'c[1,b]', 'c[1,c]']
```

5.5. Objective Function

To set the model objective equal to a linear or a quadratic expression, use the Model.setObjective() method:

```
Model.setObjective(expr, sense=GRB.MINIMIZE)
```

`expr` can be:

- LinExpr, a linear expression
- QuadExpr, a quadratic expression

`sense` is either `GRB.MINIMIZE` (the default) or `GRB.MAXIMIZE`.

```python
import gurobipy as gp
from gurobipy import GRB

import numpy as np

# Add linear objectives c^Tx
n = 5
c = np.random.rand(n)

with gp.Model("model") as model:
    x = model.addVars(n, name="x")
    linexpr = gp.quicksum(c_i * x_i for c_i, x_i in zip(c, x.values()))
    model.setObjective(linexpr)
    model.update()

    print(f"obj: {model.getObjective()}")
```

```
obj: 0.7718485205029192 x[0] + 0.14662643832254052 x[1] + 0.22459297798832945
x[2] + 0.503496718082293 x[3] + 0.743870372088844 x[4]
```

```python
import gurobipy as gp
from gurobipy import GRB

import numpy as np

n = 5
Q = np.random.rand(n, n)

with gp.Model("model") as model:
    x = model.addVars(n, name="x")
    quadexpr = 0
    # Add quadratic objective in the form x^T Q x
    for i in range(n):
        for j in range(n):
            quadexpr += x[i] * Q[i, j] * x[j]
    model.setObjective(quadexpr)
```

```python
    model.update()

    # Print objective expression
    obj = model.getObjective()
    print(f"\nobj: {obj}")
```

```
obj: 0.0 + [ 0.577773246188264 x[0] ^ 2 + 0.22815028400798743 x[0] * x[1] +
0.7057926009856951 x[0] * x[2] + 0.5668766160796567 x[0] * x[3] +
1.4647081669806783 x[0] * x[4] + 0.8558057179316467 x[1] ^ 2 +
0.8517145625303452 x[1] * x[2] + 1.0168639584110595 x[1] * x[3] +
0.7445533188159639 x[1] * x[4] + 0.8917813894437818 x[2] ^ 2 +
0.9074512745571801 x[2] * x[3] + 0.37284204762772444 x[2] * x[4] +
0.5109223586747896 x[3] ^ 2 + 1.1242190867842021 x[3] * x[4] +
0.47962043877973015 x[4] ^ 2 ]
```

5.6. Optimizing for Multiple Objectives

Gurobi supports two ways to combine multiple linear objectives:

- Blended objectives.
- Hierarchical objectives.

Objectives have:

- priorities
- weights
- absolute and relative tolerances

```
setObjectiveN(expr, index, priority=0, weight=1, abstol=1e-6, reltol=0, name='')
```

```
# Primary objective: x + 2 y
model.setObjectiveN(x + 2*y, 0, priority=0)

# Alternative, lower priority objectives: 3 y + z and x + z
model.setObjectiveN(3*y + z, 1, priority=-1)
model.setObjectiveN(x + z,   2, priority=-2)
```

5.7. SOS Constraints

A Special-Ordered Set, or SOS constraint, is a highly specialized constraint that places restrictions on the values that variables in a given list can take.

- SOS constraint of type 1 (SOS1): at most one variable is allowed to take a non-zero value.
- SOS constraint of type 2 (SOS2): at most two variables are allowed to take non-zero values, and those non-zero variables must be contiguous.

Use Model.addSOS() to add such constraints:

```
Model.addSOS(type, vars)
```

With:

- `type`: the type of SOS constraint. Can be either `GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`.
- `vars`: list of variables that participate in the consstraint.

For example, the MIP formulation of

$$z = max(x,\ y,\ 3)$$

using SOS1 constraints, is:

$$z = x + s_1 \quad (1)$$

$$z = y + s_2 \quad (2)$$

$$z = 3 + s_3 \quad (3)$$

$$v_1 + v_2 + v_3 = 1 \quad (4)$$

$$SOS1(s_1,\ v_1) \quad (5)$$

$$SOS1(s_2,\ v_2) \quad (6)$$

$$SOS1(s_3,\ v_3) \quad (7)$$

$$s_1,\ s_2,\ s_3 \in \mathbb{R}^+ \quad (8)$$

$$v_1,\ v_2,\ v_3 \in \{0,\ 1\} \quad (9)$$

5.8. General Constraints

General constraints allow you to directly model complex relationships between variables.

- Simple constraints: min, max, abs, OR, etc.

```
m.addConstr(z == gp.and_(x, y))
m.addConstr(z == gp.max_(x, y, 3))
```

- Nonlinear constraints: polynomial, exponential, logistic, trigonometric, etc

```
model.addGenConstrNL(y, nlfunc.sin(2.5 * x1) + x2)
```

```python
import gurobipy as gp
from gurobipy import nlfunc

# Minimize sin(2.5 x1) + x2
# s.t.     -1 <= x1, x2 <= 1

with gp.Env() as env, gp.Model(env=env) as model:
    x1 = model.addVar(lb=-1, ub=1, name="x1")
    x2 = model.addVar(lb=-1, ub=1, name="x2")

    y = model.addVar(lb=-float("inf"), name="y")
    model.addGenConstrNL(y, nlfunc.sin(2.5 * x1) + x2)
    model.setObjective(y)

    model.optimize()
    print(f"x1={x1.X}  x2={x2.X}  obj={y.X}")
```

```
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (mac64[arm] - Darwin 24.2.0
24C101)

CPU model: Apple M1 Pro
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 0 rows, 3 columns and 0 nonzeros
Model fingerprint: 0x00339c10
Model has 1 general nonlinear constraint (1 nonlinear terms)
Variable types: 3 continuous, 0 integer (0 binary)
Coefficient statistics:
  Matrix range     [0e+00, 0e+00]
  Objective range  [1e+00, 1e+00]
  Bounds range     [1e+00, 1e+00]
  RHS range        [0e+00, 0e+00]
Presolve model has 1 nlconstr
```

```
Added 2 variables to disaggregate expressions.
Presolve time: 0.00s
Presolved: 10 rows, 6 columns, 21 nonzeros
Presolved model has 1 nonlinear constraint(s)

Solving non-convex MINLP

Variable types: 6 continuous, 0 integer (0 binary)
Found heuristic solution: objective -2.0000000

Explored 1 nodes (0 simplex iterations) in 0.00 seconds (0.00 work units)
Thread count was 8 (of 8 available processors)

Solution count 1: -2

Optimal solution found (tolerance 1.00e-04)
Best objective -1.999999998349e+00, best bound -2.000000000000e+00, gap 0.0000%
```

```
x1=-0.6282022274684884  x2=-1.0  obj=-1.9999999983490295
```

6. Matrix-based API

- Term-based modeling can be slow
- Matrix-friendly API leans on Numpy concepts (vectorization, broadcasting)
- Model.addMVar(): Add an MVar object to a model. An `MVar` acts like a NumPy ndarray of Gurobi decision variables. An `MVar` can have an arbitrary number of dimensions, defined by the `shape` argument.

```
addMVar(shape, lb=0.0, ub=float('inf'), obj=0.0, vtype=GRB.CONTINUOUS, name='')
```

```
# A vector of 3 continuous variables
v = model.addMVar(3)
# A 5x10 matrix of binary variables
x = model.addMVar((5,10), vtype=GRB.BINARY)
```

Here's the example from the beginning of the course:

```
x = m.addMVar(shape=3, vtype=GRB.BINARY, name="x")   ①

val = np.array([1.0, 2.0, 3.0, -1.0, -1.0])
row = np.array([0, 0, 0, 1, 1])
col = np.array([0, 1, 2, 0, 1])
A = sp.csr_matrix((val, (row, col)), shape=(2, 3))   ②

rhs = np.array([4.0, -1.0])                          ③
m.addConstr(A @ x <= rhs, name="c")                  ④

obj = np.array([1.0, 1.0, 2.0])
m.setObjective(obj @ x, GRB.MAXIMIZE)                ⑤
```

① Create variables

② Build a (sparse) matrix

③ Build an rhs vector

④ Add constraints

⑤ Set objective

Here is another with timing of the difference.

```python
import gurobipy as gp
import numpy as np
from timeit import default_timer

n = 1000
Q = np.random.rand(n, n)

def term_based():
    with gp.Model("term-based") as model:
        x = model.addVars(n, name="x")
        model.addConstr(
            gp.quicksum(x[i] * Q[i, j] * x[j] for j in range(n)
                                             for i in range(n)) <= 10
        )
```

```python
def matrix_api():
    with gp.Model("matrix-based") as model:
        x = model.addMVar(n, name="x")
        model.addConstr(x.T @ Q @ x <= 10)

matrix_api()  # To create the default env
for f in [term_based, matrix_api]:
    start = default_timer()
    f()
    end = default_timer()
    print(f"Running {f.__name__} took {end - start} seconds")
```

```
Running term_based took 4.9921418750309385 seconds
Running matrix_api took 0.1097855000407435 seconds
```

7. Interacting with the Model

7.1. Attributes

The primary mechanism for querying and modifying properties of a Gurobi object is through the attribute interface. Attributes exist on instances of Model, Variable, all types of constraints, and more.

Model:

- number of modeling elements of each type (NumConstrs, NumVars, etc.)
- information about the type of model (IsMip, IsMultiObj, etc.), its statistics (SolCount, NodeCount, etc.)
- information about the solutions found (SolCount), the best known bound (ObjBound), the gap (MipGap), etc.
- ...

Variables:

- lower (LB) and upper (UB) bounds
- value in a MIP start vector (Start)
- value in the best solution (X)
- ...

Constraints:

- right-hand side value (RHS)
- dual value in the best solution (Pi)
- ...

```python
import gurobipy as gp
from gurobipy import GRB

with gp.read("data/glass4.mps.bz2") as model:
    model.optimize()

    print("****************** SOLUTION ******************")
    print(f"\tStatus       : {model.Status}")
    print(f"\tObj          : {model.ObjVal}")
    print(f"\tSolutionCount: {model.SolCount}")
    print(f"\tRuntime      : {model.Runtime}")
    print(f"\tMIPGap       : {model.MIPGap}")

    print("\n")
    for var in model.getVars()[:20]:
        print(f"\t{var.VarName} = {var.X}")
```

```
Read MPS format model from file data/glass4.mps.bz2
Reading time = 0.01 seconds
glass4: 396 rows, 322 columns, 1815 nonzeros
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (mac64[arm] - Darwin 24.2.0
24C101)

CPU model: Apple M1 Pro
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 396 rows, 322 columns and 1815 nonzeros
Model fingerprint: 0x18b19fdf
Variable types: 20 continuous, 302 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 8e+06]
  Objective range  [1e+00, 1e+06]
  Bounds range     [1e+00, 8e+02]
```

```
  RHS range        [1e+00, 8e+06]
Presolve removed 6 rows and 6 columns
Presolve time: 0.00s
Presolved: 390 rows, 316 columns, 1803 nonzeros
Variable types: 19 continuous, 297 integer (297 binary)
Found heuristic solution: objective 3.133356e+09

Root relaxation: objective 8.000024e+08, 72 iterations, 0.00 seconds (0.00 work
units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time


     0     0 8.0000e+08    0   72 3.1334e+09 8.0000e+08  74.5%     -    0s
H    0     0                    2.600019e+09 8.0000e+08  69.2%     -    0s
H    0     0                    2.366684e+09 8.0000e+08  66.2%     -    0s
     0     0 8.0000e+08    0   72 2.3667e+09 8.0000e+08  66.2%     -    0s
```

```
     0     0 8.0000e+08     0   72 2.3667e+09 8.0000e+08  66.2%     -    0s
     0     0 8.0000e+08     0   77 2.3667e+09 8.0000e+08  66.2%     -    0s
     0     0 8.0000e+08     0   76 2.3667e+09 8.0000e+08  66.2%     -    0s
     0     2 8.0000e+08     0   75 2.3667e+09 8.0000e+08  66.2%     -    0s
H   26    80                    2.116683e+09 8.0000e+08  62.2%  38.4    0s
H   36    80                    2.016683e+09 8.0000e+08  60.3%  30.6    0s
H   65    80                    2.000015e+09 8.0000e+08  60.0%  20.1    0s
H  251   274                    1.991127e+09 8.0000e+08  59.8%  10.8    0s
* 1128  1027             112    1.920014e+09 8.0000e+08  58.3%   6.3    0s
H 1695  1546                    1.812517e+09 8.0000e+08  55.9%   6.2    0s
H 1788  1729                    1.800017e+09 8.0000e+08  55.6%   6.1    0s
H 2010  1645                    1.766684e+09 8.0000e+08  54.7%   6.0    0s
H 2353  1796                    1.700017e+09 8.0000e+08  52.9%   5.6    0s
H 2399  1782                    1.700017e+09 8.0000e+08  52.9%   5.6    0s
H 3109  1938                    1.700016e+09 8.0000e+08  52.9%   5.4    0s
* 9626  5603              77    1.700016e+09 8.0000e+08  52.9%   3.9    0s
* 9627  5603              77    1.700016e+09 8.0000e+08  52.9%   3.9    0s
```

```
H10514  6175                               1.700016e+09 8.0000e+08  52.9%   3.9    0s
H11625  7019                               1.666683e+09 8.0000e+08  52.0%   3.8    0s
H11975  6974                               1.650016e+09 8.0000e+08  51.5%   3.7    0s
H13781  7848                               1.650016e+09 8.0000e+08  51.5%   3.6    0s
H13830  7848                               1.650016e+09 8.0000e+08  51.5%   3.6    0s
H14108  8324                               1.600015e+09 8.0000e+08  50.0%   3.6    0s
H16117  9426                               1.600015e+09 8.0000e+08  50.0%   3.5    0s
H18938 10840                               1.600014e+09 8.0000e+08  50.0%   3.5    1s
H19146  9570                               1.500013e+09 8.0000e+08  46.7%   3.5    1s
H20592 10593                               1.500013e+09 8.0000e+08  46.7%   3.5    1s
H30377 13890                               1.500012e+09 8.3576e+08  44.3%   3.7    2s
 30568 14024 1.1000e+09   66   89 1.5000e+09 8.8699e+08  40.9%   3.9    5s
 30880 14242 1.0000e+09   27  114 1.5000e+09 8.9978e+08  40.0%   4.3   10s
 39957 16368 1.0000e+09  177   54 1.5000e+09 9.2005e+08  38.7%   6.4   15s
*135241 22495             219    1.400016e+09 1.1000e+09  21.4%   7.0   19s
*136789 22885             210    1.400014e+09 1.1000e+09  21.4%   7.0   19s
*142250 22031             209    1.400013e+09 1.1098e+09  20.7%   7.0   19s
```

```
 148470 22002 1.4000e+09  186   35 1.4000e+09 1.1750e+09  16.1%   7.0   20s
*186816 33080             211    1.400013e+09 1.2000e+09  14.3%   7.1   21s
*228748 44827             206    1.400013e+09 1.2000e+09  14.3%   6.9   23s
 266386 52292 infeasible  203      1.4000e+09 1.2000e+09  14.3%   6.7   25s
*284522  5732             204    1.200013e+09 1.2000e+09  0.00%   6.6   25s

Cutting planes:
  Learned: 1
  Gomory: 7
  Implied bound: 12
  Projected implied bound: 1
  MIR: 42
  Flow cover: 17
  RLT: 5
  Relax-and-lift: 16

Explored 284537 nodes (1881113 simplex iterations) in 25.78 seconds (29.59 work
```

```
units)
Thread count was 8 (of 8 available processors)

Solution count 10: 1.20001e+09 1.40001e+09 1.40001e+09 ... 1.60001e+09

Optimal solution found (tolerance 1.00e-04)
Best objective 1.200012600000e+09, best bound 1.200009038285e+09, gap 0.0003%
****************** SOLUTION ******************
    Status       : 2
    Obj          : 1200012600.0
    SolutionCount: 10
    Runtime      : 25.77737784385681
    MIPGap       : 2.968064893289939e-06



    x1 = 0.0
    x2 = 700.0000000000001
```

```
    x3 = 1000.0
    x4 = 1000.0
    x5 = 400.0
    x6 = 200.0
    x7 = 200.0
    x8 = 500.0
    x9 = 700.0
    x10 = 1200.0
    x11 = 0.0
    x12 = 0.0
    x13 = 200.0
    x14 = 800.0
    x15 = 800.0
    x16 = 600.0
    x17 = 300.0
    x18 = 300.0
    x19 = 199.99999999999991
```

```
    x20 = 1.0
```

7.2. Parameters

Parameters control the mechanics of the Gurobi Optimizer.

```python
import gurobipy as gp
from gurobipy import GRB

with gp.read("data/glass4.mps.bz2") as model:
    model.params.Threads = 1
    model.params.TimeLimit = 10
    model.optimize()
```

```
Read MPS format model from file data/glass4.mps.bz2
Reading time = 0.02 seconds
glass4: 396 rows, 322 columns, 1815 nonzeros
Set parameter Threads to value 1
Set parameter TimeLimit to value 10
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (mac64[arm] - Darwin 24.2.0
24C101)

CPU model: Apple M1 Pro
Thread count: 8 physical cores, 8 logical processors, using up to 1 threads

Non-default parameters:
TimeLimit  10
Threads  1

Optimize a model with 396 rows, 322 columns and 1815 nonzeros
```

```
Model fingerprint: 0x18b19fdf
Variable types: 20 continuous, 302 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 8e+06]
  Objective range  [1e+00, 1e+06]
  Bounds range     [1e+00, 8e+02]
  RHS range        [1e+00, 8e+06]
Presolve removed 6 rows and 6 columns
Presolve time: 0.00s
Presolved: 390 rows, 316 columns, 1803 nonzeros
Variable types: 19 continuous, 297 integer (297 binary)
Found heuristic solution: objective 3.133356e+09

Root relaxation: objective 8.000024e+08, 72 iterations, 0.00 seconds (0.00 work
units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
```

```
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time

     0     0 8.0000e+08    0   72 3.1334e+09 8.0000e+08  74.5%     -    0s
H    0     0                    2.600019e+09 8.0000e+08  69.2%     -    0s
     0     0 8.0000e+08    0   72 2.6000e+09 8.0000e+08  69.2%     -    0s
     0     0 8.0000e+08    0   72 2.6000e+09 8.0000e+08  69.2%     -    0s
     0     0 8.0000e+08    0   76 2.6000e+09 8.0000e+08  69.2%     -    0s
     0     0 8.0000e+08    0   76 2.6000e+09 8.0000e+08  69.2%     -    0s
H    0     0                    2.500018e+09 8.0000e+08  68.0%     -    0s
H    0     0                    2.400019e+09 8.0000e+08  66.7%     -    0s
     0     2 8.0000e+08    0   76 2.4000e+09 8.0000e+08  66.7%     -    0s
H   52    52                    2.288909e+09 8.0000e+08  65.0%   4.1    0s
H   52    52                    2.200018e+09 8.0000e+08  63.6%   4.1    0s
H   52    52                    2.150019e+09 8.0000e+08  62.8%   4.1    0s
H   52    52                    2.133352e+09 8.0000e+08  62.5%   4.1    0s
H   54    54                    2.000018e+09 8.0000e+08  60.0%   4.1    0s
H  461   419                    2.000018e+09 8.0000e+08  60.0%   4.1    0s
```

```
H  461   419                         2.000017e+09 8.0000e+08  60.0%   4.1    0s
H  461   411                         1.950017e+09 8.0000e+08  59.0%   4.1    0s
H  461   411                         1.933350e+09 8.0000e+08  58.6%   4.1    0s
H  461   411                         1.900017e+09 8.0000e+08  57.9%   4.1    0s
H  547   479                         1.900017e+09 8.0000e+08  57.9%   4.3    0s
H  688   536                         1.900017e+09 8.0000e+08  57.9%   4.4    0s
H  708   526                         1.900016e+09 8.0000e+08  57.9%   4.4    0s
H  708   501                         1.866683e+09 8.0000e+08  57.1%   4.4    0s
*  735   483                  126    1.833351e+09 8.0000e+08  56.4%   4.5    0s
H  931   559                         1.833351e+09 8.0000e+08  56.4%   4.4    0s
H  931   538                         1.800017e+09 8.0000e+08  55.6%   4.4    0s
H  931   520                         1.800017e+09 8.0000e+08  55.6%   4.4    0s
H  958   518                         1.800017e+09 8.0000e+08  55.6%   4.4    0s
H  958   502                         1.800017e+09 8.0000e+08  55.6%   4.4    0s
* 1108   537                   83    1.800015e+09 8.0000e+08  55.6%   4.6    0s
H 2106  1138                         1.775015e+09 8.0000e+08  54.9%   4.1    0s
H 2555  1485                         1.775015e+09 8.0000e+08  54.9%   4.0    0s
```

```
* 4189  2637                57    1.758351e+09 8.0000e+08  54.5%   3.9    1s
H 4665  2932                      1.725017e+09 8.0000e+08  53.6%   3.9    1s
H 5665  3637                      1.725017e+09 8.0000e+08  53.6%   3.9    1s
H 5665  3621                      1.700016e+09 8.0000e+08  52.9%   3.9    1s
H 6130  3954                      1.700016e+09 8.0000e+08  52.9%   3.9    1s
 10353  6940 1.2200e+09   56  123 1.7000e+09 8.4590e+08  50.2%   4.3    5s
H10418  6636                      1.675016e+09 8.4985e+08  49.3%   4.4    5s
H10435  6314                      1.650016e+09 8.5187e+08  48.4%   4.4    6s
H10435  5998                      1.650016e+09 8.5187e+08  48.4%   4.4    6s
H10760  5883                      1.600016e+09 8.8011e+08  45.0%   5.1    8s
H10801  5616                      1.600016e+09 8.8011e+08  45.0%   5.2    8s
H10801  5339                      1.600016e+09 8.8011e+08  45.0%   5.2    8s
H11173  5253                      1.600016e+09 9.0000e+08  43.8%   5.7    9s

Cutting planes:
  Learned: 1
  Gomory: 11
```

```
  Cover: 1
  Implied bound: 5
  Projected implied bound: 2
  Clique: 1
  MIR: 34
  Flow cover: 20
  RLT: 5
  Relax-and-lift: 13

Explored 12753 nodes (79852 simplex iterations) in 10.00 seconds (11.07 work
units)
Thread count was 1 (of 8 available processors)

Solution count 10: 1.60002e+09 1.60002e+09 1.60002e+09 ... 1.72502e+09

Time limit reached
```

```
Best objective 1.600015500000e+09, best bound 9.000054810120e+08, gap 43.7502%
```

7.3. Callbacks

A callback is a user-defined function invoked by Gurobi while the optimization process is going on. They enable more control over the optimization. They can be used to:

- customize the termination of the solve
- add user cuts and lazy constraints
- add custom feasible solutions
- monitor the progress of optimization
- customize the optimization progress display

A callback must be a function (actually, any callable object) that accepts two arguments:

- `model`: the model that is being solved
- `where`: from where is the Gurobi Optimizer is the callback invoked (presolve, simplex, barrier, MIP, at a node, etc.)

A callback can query information from the solver using Model.cbGet(). The type of information that can be queried depends on from `where` the callback is invoked. For example, when `where == PRESOLVE`, you can query the number of rows removed using `what == PRE_ROWDEL`. All the callback code values that you can query are listed here.

```
import gurobipy as gp
from gurobipy import GRB
from functools import partial

# Used by the callback to store or retrieve information
class CallbackData:
    def __init__(self):
        self.invocations = 0

# The callback, the function that will be invoked
def mycallback(model, where, *, cbdata):
    # We're only interested in this case
    if where == GRB.Callback.MIP:
        cbdata.invocations += 1
```

```python
        # Get information from the model
        nodecnt = model.cbGet(GRB.Callback.MIP_NODCNT)
        if nodecnt > 100:
            # Terminate the search
            model.terminate()



# Create a model from an instance file
with gp.read("data/glass4.mps.bz2") as model:
    # Create the object to store/retrieve information
    cbdata = CallbackData()
    # Create a callable with two args: model and where
    callback_func = partial(mycallback, cbdata=cbdata)

    # Optimize, with the callback
    model.optimize(callback_func)
```

```
    # Confirm the callback was invoked
    print(f"MIP Callback was invoked {cbdata.invocations} times.")
```

```
Read MPS format model from file data/glass4.mps.bz2
Reading time = 0.03 seconds
glass4: 396 rows, 322 columns, 1815 nonzeros
Gurobi Optimizer version 12.0.0 build v12.0.0rc1 (mac64[arm] - Darwin 24.2.0
24C101)

CPU model: Apple M1 Pro
Thread count: 8 physical cores, 8 logical processors, using up to 8 threads

Optimize a model with 396 rows, 322 columns and 1815 nonzeros
Model fingerprint: 0x18b19fdf
Variable types: 20 continuous, 302 integer (0 binary)
Coefficient statistics:
  Matrix range     [1e+00, 8e+06]
  Objective range  [1e+00, 1e+06]
  Bounds range     [1e+00, 8e+02]
```

```
  RHS range        [1e+00, 8e+06]
Presolve removed 6 rows and 6 columns
Presolve time: 0.00s
Presolved: 390 rows, 316 columns, 1803 nonzeros
Variable types: 19 continuous, 297 integer (297 binary)
Found heuristic solution: objective 3.133356e+09

Root relaxation: objective 8.000024e+08, 72 iterations, 0.00 seconds (0.00 work
units)

    Nodes    |    Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |  Obj  Depth IntInf | Incumbent    BestBd   Gap | It/Node Time


     0     0 8.0000e+08    0   72 3.1334e+09 8.0000e+08  74.5%     -    0s
H    0     0                    2.600019e+09 8.0000e+08  69.2%     -    0s
H    0     0                    2.366684e+09 8.0000e+08  66.2%     -    0s
     0     0 8.0000e+08    0   72 2.3667e+09 8.0000e+08  66.2%     -    0s
```

```
     0     0 8.0000e+08    0   72 2.3667e+09 8.0000e+08  66.2%     -    0s
     0     0 8.0000e+08    0   77 2.3667e+09 8.0000e+08  66.2%     -    0s
     0     0 8.0000e+08    0   76 2.3667e+09 8.0000e+08  66.2%     -    0s
     0     2 8.0000e+08    0   75 2.3667e+09 8.0000e+08  66.2%     -    0s
H   26    80                     2.116683e+09 8.0000e+08  62.2%  38.4    0s
H   36    80                     2.016683e+09 8.0000e+08  60.3%  30.6    0s
H   65    80                     2.000015e+09 8.0000e+08  60.0%  20.1    0s

Cutting planes:
  Gomory: 86
  Cover: 2
  Implied bound: 31
  MIR: 2
  RLT: 70
  Relax-and-lift: 40
  PSD: 7
```

```
Explored 161 nodes (2760 simplex iterations) in 0.09 seconds (0.08 work units)
Thread count was 8 (of 8 available processors)

Solution count 6: 2.00002e+09 2.01668e+09 2.11668e+09 ... 3.13336e+09

Solve interrupted
Best objective 2.000015200000e+09, best bound 8.000033092112e+08, gap 60.0001%

User-callback calls 606, time in user-callback 0.00 sec
MIP Callback was invoked 117 times.
```

Other methods of interest are:

- Model.cbGetNodeRel(): retrieve the values of the variables in the node relaxation solution at the current node.
- Model.cbGetSolution(): retrieve the values of the variables in the new MIP solution.
- Model.cbCut(): add a new cutting plane to the model.
- Model.cbLazy(): add a new lazy constraint to the model.
- Model.cbSetSolution(): import a user-constructed solution into Gurobi.

8. Requirements and installation

8.1. Setup for Windows

8.1.1. Check for Anaconda 3.9 to 3.12

If you have Anaconda Python installed and this is version 3.9, 3.10, 3.11 or 3.12, just go ahead and run

```
conda install gurobipy
```

You can now go directly to section Verification of the installation.

8.1.2. Check for the `py` launcher

Check whether you already have a 'core' Python version installed:

- open a Command Prompt and run

```
py --list
```

- If `py --list` tells you that you have version 3.9, 3.10, 3.11 or 3.12, you can go ahead to the Installation of Gurobipy section below.

- If you get

```
'py' is not recognized as an internal or external command,
operable program or batch file.
```

then you need to install Python 3.12, as per the next section.

8.1.3. Install Python 3.12

If you have none of 3.9, 3.10, 3.11 or 3.12, install the latest 3.12 Python version from https://www.python.org/downloads/windows/.

- Download the 'Windows installer (64-bit)', unless you have an ARM machine
- Run the installer (you don’t need to do it as an Administrator)
- UNCHECK 'Install launcher for all users'
- Make sure that 'Add Python 3.12 to the path' is NOT checked (so that your access to your existing Python version is not

changed)

- open a Command Prompt and run

```
py --list
```

- Verify that you have 3.12 listed

For more details on the installation process, refer to https://docs.python.org/3.12/using/windows.html.

8.1.4. Create a venv and activate it

The reason to create a `venv` is to fix the Python version and isolate the work done in the venv to make sure it doesn't impact the rest of your system.

- In a Command Prompt, run

```
md gurobipy-course
cd gurobipy-course
py -3.12 -m venv venv --prompt "gurobipy-course"
```

Now that the venv has been created, you can activate it with the command below. Note that you should always do this in any new

Command Prompt window that you start.

```
venv/Scripts/Activate
```

You can now continue in Installation of Gurobipy

8.2. Setup for macOS

- I will assume here that you have already installed Homebrew. Otherwise, follow the instructions there.
- Install `pyenv` and Python 3.12

```
brew install pyenv
pyenv install 3.12
```

- Create a local directory for the course.

```
mkdir gurobipy-course
cd gurobipy-course
```

- Configure pyenv to use Python 3.12 when in this directory.

```
pyenv local 3.12
```

- Create a virtual environmnent for the course and activate it. From now on, remember to always activate your venv in whatever shell you use.

```
python -m venv venv --prompt "gurobipy-course"
source venv/bin/activate
```

- I also highly recommend that you set the variable `PIP_REQUIRE_VIRTUALENV=true` in your environment. This forces `pip` to fail when not running in a virtual environment.

8.3. Installation of Gurobipy

Confirm that your venv is active: you should see that your prompt is modified to mention 'gurobipy-course'. With your venv activated, confirm that

```
python --version
```

runs Python 3.12

You can now run

```
pip install gurobipy
```

8.4. Verification of the installation

You can check that gurobipy is correctly installed by running

```
python -c "import gurobipy as gp; m = gp.Model(); print(gp.GRB.VERSION_MAJOR)"
```

This should output one or a few lines about your license, then the Gurobi version number, which should be 12 at this time. An example could be:

```
Restricted license - for non-production use only - expires 2026-11-23
12
```

8.5. Get a free Gurobi license

- Using your academic email address, create an account on https://portal.gurobi.com.
- **From a computer that belongs to your University**, log onto https://portal.gurobi.com/iam/licenses/request?type=academic and request a 'WLS Academic' license
- Once you see your license at https://portal.gurobi.com/iam/licenses/list, open https://license.gurobi.com/manager/licenses/, click 'Download' at the bottom of the license card, give a name and a description to the API key that you’re about to create.

- When the 'API key created' dialog appears, click the 'Download' button and save the 'gurobi.lic' file.
- Copy that file to your laptop, in your user profile folder (typically `C:\Users\[name]`)
- Verify that running

```
python -c "import gurobipy as gp; m = gp.Model()"
```

does **not** give you any more the 'Restricted license' message.

For more details, refer to this page.