Gurobi Python Interface: Matrix-friendly Modeling Techniques

GUROBI OPTIMIZATION

The World's Fastest Solver

Speaker introduction: Dr. Robert Luce

- Member of Gurobi's development team
- Experienced researcher in applied mathematics
- Research interests: numerical linear algebra, numerical optimization and applied function theory
- Graduated from Technical University of Berlin

gurobipy overview

- Our Python interface for Gurobi.
- Lightweight modeling objects for variables, constraints, etc.
- Syntactic sugar for modeling through operators and rich comparisons.

```
import gurobipy as gp

m = gp.Model()
x = m.addVar()
m.addConstr(x >= 42)
```

- Quick start instructions to run examples:
  - Go to the Gurobi installation directory (“GUROBI_HOME”)
  - `python setup.py install`
  - `pip install numpy scipy`

gurobipy becomes more matrix-friendly!

GUROBI OPTIMIZATION

- Numpy ndarray's are ubiquitous.
- Sparse matrices are widespread, too (through Scipy.sparse).
- Gurobi version 9.0 has greatly improved modeling capabilities with such data structures.
- If you run examples from these slides, please always have the following imported:
  - `import numpy as np`
  - `import scipy.sparse as sp`
  - `import gurobipy as gp`
- For conciseness these import statements are often omitted from the examples shown here.

“Term-based” modeling

Build optimization model by composing linear and quadratic expressions by terms:

```
m = gp.Model() # Create an optimization model

x = m.addVar(ub=1.0) # Create three variables in [0,1]
y = m.addVar(ub=1.0)
z = m.addVar(ub=1.0)

m.setObjective(x*x + 2*y*y + 3*z*z) # A quadratic objective function

m.addConstr(x + 2*y + 3*z >= 4) # Two linear constraints
m.addConstr(x + y >= 1)
```

What if your optimization data is naturally expressed in terms of matrices and vectors?

GUROBI
OPTIMIZATION

$$
\begin{aligned}
\min_x \quad & x^T Q x \\
\text{s.t.} \quad & Ax \geq b \\
& x \geq 0
\end{aligned}
$$

- We would need to traverse nonzeros of Q and A.
- We would need to construct explicit modeling objects for all the expressions.
- Somewhat superfluous since these expressions are already defined through the matrix-vector relations between Q, A, x and b on a higher syntactic level.

New in 9.0: You can build optimization models directly in terms of matrices and vectors.

- Exemplary use cases:
  - A is a given node-arc incidence matrix, and we want want to express flow conservation Ax = f.
  - Q is a given covariance matrix, and we want to minimize variance.

Our example rewritten with matrix data

```
m = gp.Model()

x = m.addVar(ub=1.0)
y = m.addVar(ub=1.0)
z = m.addVar(ub=1.0)

m.setObjective(x*x + 2*y*y + 3*z*z)

m.addConstr(x + 2*y + 3*z >= 4)
m.addConstr(x + y >= 1)
```

```
Q = np.diag([1, 2, 3])
A = np.array([ [1, 2, 3], [1, 1, 0] ])
b = np.array([4, 1])

m = gp.Model()

x = m.addMVar(3, ub=1.0)
m.setObjective(x @ Q @ x)
m.addConstr(A@x >= b)
```

Matrix variables: `MVar` objects

- An object representing a vector (or matrix) of optimization variables.
- Constructed by the factory function `gp.Model.addMVar`.

```
def Model.addMVar(shape, lb=0, ub=float('inf'), obj=0, vtype='C', name=""):
```

- `shape`: tuple of dimensions of the variable (like for Numpy's `ndarray`).
- `lb`: lower bound(s) of the variable.
- ub: upper bound(s) of the variable.
- obj: objective coefficient(s).
- vtype: variable type(s) (continuous, binary, etc.).

All kwargs can be iterables; scalar arguments are broadcast!

MVar construction examples

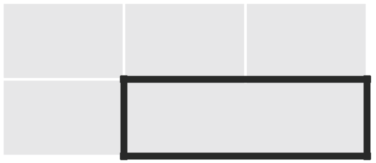
```
# Add a 4-by-2 matrix binary variable
model.addMVar((4,2), vtype=GRB.BINARY)

# Add a vector of three variables with non-default lower bounds
model.addMVar((3,), lb=[-1, -2, -1]) # lb is an iterable

# Same as above: 1-D shape tuples don't need to be spelled out
model.addMVar(3, lb=[-1, -2, -1])

# Add a 8-by-8-by-8-by-8 four-dimensional variable
model.addMVar((8, 8, 8, 8))
```

Useful operations with MVars



```
>>> model = gp.Model(); model.setParam('OutputFlag', 0)
>>> x = model.addMVar((2,3), ub=range(6), obj=1); model.update()
>>> x[1, 1:3] # slicing
<(2,) matrix variable>

>>> x.UB # Query Gurobi attribute, gives ndarray!
array([[0., 1., 2.],
       [3., 4., 5.]])

>>> x[:, 2].obj = -1.0 # Set Gurobi attribute on slice via broadcast
>>> model.update(); x.obj # Did it work? (Yes.)
array([[ 1.,  1., -1.],
       [ 1.,  1., -1.]])

>>> model.optimize() # Solve, default optimization sense is ‘minimize’
>>> x.X # Get solution values as ndarray
array([[0., 0., 2.],
       [0., 0., 5.]])
```

Building linear constraints with MVars

- **Use Python3 matrix multiplication operator `@` to build linear expressions and constraints.**
- **Typical usage pattern:** `model.addConstr(A @ x == b)`
  - A is a Numpy ndarray, or a Scipy.sparse matrix.
  - b is a Numpy ndarray.
  - (the senses <= and >= can be used just as well).
- **Example: Random sparse linear system**

```
# Data: 8 equations, 128 variables, 20% density
A = sp.random(8, 128, 0.2)
b = np.random.rand(8)

# Build optimization model
model = gp.Model()
x = model.addMVar(128)
model.addConstr(A@x == b)
```

Further examples of creating linear constraints

GUROBI OPTIMIZATION

```
# Some data to play with
A = np.random.rand(5,3)
D = np.random.rand(5,10)

model = gp.Model()
x = model.addMVar(3)
model.addConstr(A @ x == 1) # RHS is broadcast!

y = model.addMVar(5)
model.addConstr(y == A @ x) # y = Ax
model.addConstr(y.sum() <= 0.5) # Mvar.sum() is for convenience

z = model.addMVar(10)
model.addConstr(A @ x + D @ z == 0) # Column wise composition
```

Caveat: Linear constraints are 1-D

- Altough `MVar` objects can be N-D, linear constraints involving `MVars` are restricted to one dimension.
- Example:

```
A = np.random.rand(4,4)
B = np.random.rand(4,4)
model = gp.Model()
X = model.addMVar((4,4))
# Not (yet) supported: Add 16 linear constraints AX = B
model.addConstr(A@X == B) # Error!

# Instead: Add four 4-by-1 constraints
model.addConstrs(A@X[:, j] == B[:, j] for j in range(4))
```

Quadratic expressions and constraints

- **Syntactically: Use matrix multiplication (surprise!)**
- **Concrete examples:**

```
model = gp.Model()
x = model.addMVar(3)
model.setObjective(x @ x) # x[0]^2 + x[1]^2 + x[2]^2

F = np.random.rand(100, 10) # Think of a time series, 10 assets
y = model.addMVar(10)
Sigma = F.T @ F # Covariance matrix
model.setObjective(y @ Sigma @ y)

s = model.addMVar(100)
z = model.addMVar(100)
model.addConstr(s @ z == 0) # Complementarity constraint, nonconvex!
```

Performance considerations

```
m = gp.Model()

x = m.addVar(ub=1.0)
y = m.addVar(ub=1.0)
z = m.addVar(ub=1.0)

m.setObjective(x*x + 2*y*y + 3*z*z)

m.addConstr(x + 2*y + 3*z >= 4)
m.addConstr(x + y >= 1)
```

- Each operator application + and ∗ results in some Python object manipulations.
- Overhead usually low, but it can become expensive sometimes.

```
Q = np.diag([1, 2, 3])
A = np.array([ [1, 2, 3], [1, 1, 0] ])
b = np.array([4, 1])

m = gp.Model()

x = m.addMVar(3, ub=1.0)
m.setObjective(x @ Q @ x)
m.addConstr(A@x >= b)
```

- No elementary operations needed, everything is expressed by @
- The data A, Q, b is only read on the C level, no intermediate arithmetic on terms.

Benchmark example: Three ways to model a random sparse linear system

```
# Generate data for Ax=b, x>=0
m = 1024; n = 8192; d = 0.2
A = sp.random(m, n, d, format='csr')
b = np.random.rand(m)


# Run the three code snippets here
# [...]
MVar  time: 0.083sec.
Term1 time: 1.729sec.
Term2 time: 53.77sec.
```

With MVar

```
x = model.addMVar(n)
model.addConstr(A@x == b)
```

Best(?) w/o MVar

```
x = model.addVars(n).values()
for i in range(m):
    (_, colidx, colcoef) = sp.find(A[i, :])
    le = gp.LinExpr(colcoef, [x[j] for j in colidx])
    model.addConstr(le == b[i])
```

Naive approach

```
(rowidx, colidx, coef) = sp.find(A)
x = model.addVars(n).values()
le = [gp.LinExpr() for i in range(m)]
for k in range(len(coef)):
    le[rowidx[k]] += coef[k] * x[colidx[k]]
for i in range(m):
    model.addConstr(le[i] == b[i])
```

Bare metal API functions

- Also new in Gurobi 9.0: A lower level interface for modeling with matrix data.
- No modeling objects involved at all, almost no overhead
- Add linear constraints
  - `Model.addMConstrs(A, x, sense, b)`
- Add quadratic constraints
  - `Model.addMQConstr(Q, c, sense, rhs, xQ_L=None, xQ_R=None, xc=None)`
- Set quadratic and linear objective functions
  - `Model.setMObjective(Q, c, constant, xQ_L=None, xQ_R=None, xc=None, sense=None)`
- These API functions just take the raw data, and possibly a subset of variables.
- Useful in certain specialized situations, or if one is tied to Python 2.

What we have seen, and what we will see

- If your optimization data is naturally expressed in matrices and vectors, gurobipy 9.0 gives you tools to work more idiomatically with that data.
- This is our first step towards making guropbipy matrix-friendly – let us know what you would like to see next!

Thanks!

Your Next Steps

- Try Gurobi 9.0 Now!
    - Get a 30-day commercial trial license of Gurobi at www.gurobi.com/free-trial
    - Academic and research licenses are free.
- For questions about Gurobi pricing, please contact sales@gurobi.com or sales@gurobi.de
- A recording of this webinar, including the slides, will be available in roughly one week.