Modeling I

Anwendertage 2017

Frankfurt, Germany

Agenda for this session

- Small demos

- Useful knowledge
  - Gurobi model components
  - What makes a model difficult?
  - Choosing an interface
  - Programming pitfalls
  - Model debugging

Gurobi model components

- Decision variables
- Objective function
  - minimize $\mathbf{x}^T\mathbf{Q}\mathbf{x} + \mathbf{c}^T\mathbf{x} + \alpha$
- Constraints
  - $\mathbf{A}\mathbf{x} = \mathbf{b}$ (linear constraints)
  - $\mathbf{l} \leq \mathbf{x} \leq \mathbf{u}$ (bound constraints)
  - some x_h integral (integrality constraints)
  - some x_i lie within second order cones (cone constraints)
  - $\mathbf{x}^T\mathbf{Q}_j\mathbf{x} + \mathbf{q}_j^T\mathbf{x} \leq \beta_j$ (quadratic constraints)
  - some x_k in SOS (special ordered set constraints)

- Many of these are optional

Example – Mixed Integer Linear Program (MILP)

- Decision variables
- Objective function
  - minimize $\mathbf{x}^T\mathbf{Q}\mathbf{x}$ + $\mathbf{c}^T\mathbf{x}$ + α
- Constraints
  - $\mathbf{A}\mathbf{x} = \mathbf{b}$ (linear constraints)
  - $\mathbf{l} \le \mathbf{x} \le \mathbf{u}$ (bound constraints)
  - some x_h integral (integrality constraints)
  - some x_i lie within second order cones (cone constraints)
  - $\mathbf{x}^T\mathbf{Q}_j\mathbf{x} + \mathbf{q}_j^T\mathbf{x} \le \beta_j$ (quadratic constraints)
  - some x_k in SOS (special ordered set constraints)

- By far, most common model for Gurobi users

MIP is versatile

- Giant leap from linear programming (LP) with respect to modeling power
  - Modeling with MIP is more than LP with integer restrictions

- MIP versatility typically comes from binary decision variables
  - b_k = 0/1
  - Captures yes/no decisions

- Combine with linear constraints to capture complex relationships between decisions
  - Ex: fixed charge for using a resource

    minimize … + 100 b_k + …

    subject to $x_k \leq 10\ b_k$
  - Ex: pick one from among a set of options

    $b_1 + b_2 + b_3 = 1$
  - …

Industries using Gurobi

- Accounting
- Advertising
- Agriculture
- Airlines
- ATM provisioning
- Compilers
- Defense
- Electrical power
- Energy
- Finance
- Food service
- Forestry
- Gas distribution
- Government
- Internet applications
- Logistics/supply chain
- Medical
- Mining
- National research labs
- Online dating
- Portfolio management
- Railways
- Recycling
- Revenue management
- Semiconductor
- Shipping
- Social networking
- Sourcing
- Sports betting
- Sports scheduling
- Statistics
- Steel manufacturing
- Telecommunications
- Transportation
- Utilities
- Workforce scheduling

Creating and Solving Your First Model #1

- Simple example:
  - You want to decide about three activities (do or don‘t do) and aim for maximum value
  - You need to choose at least activity 1 or 2 (or both)
  - The total time limit is 4 hours
    - Activity 1 takes 1 hours
    - Activity 2 takes 2 hours
    - Activity 3 takes 4 hours
  - Activity 3 is worth twice as much as 1 and 2

$$\begin{aligned} \max \quad & x + y + 2z \\ \text{s.t.} \quad & x + 2y + 4z \leq 4 \\ & x + y \geq 1 \\ & x, y, z \in \{0, 1\} \end{aligned}$$

- This can be modeled as a mixed-integer linear program
  - Binary variables x,y,z for activities 1,2,3
  - Linear constraint for time limit
  - Linear constraint for condition (1 or 2)

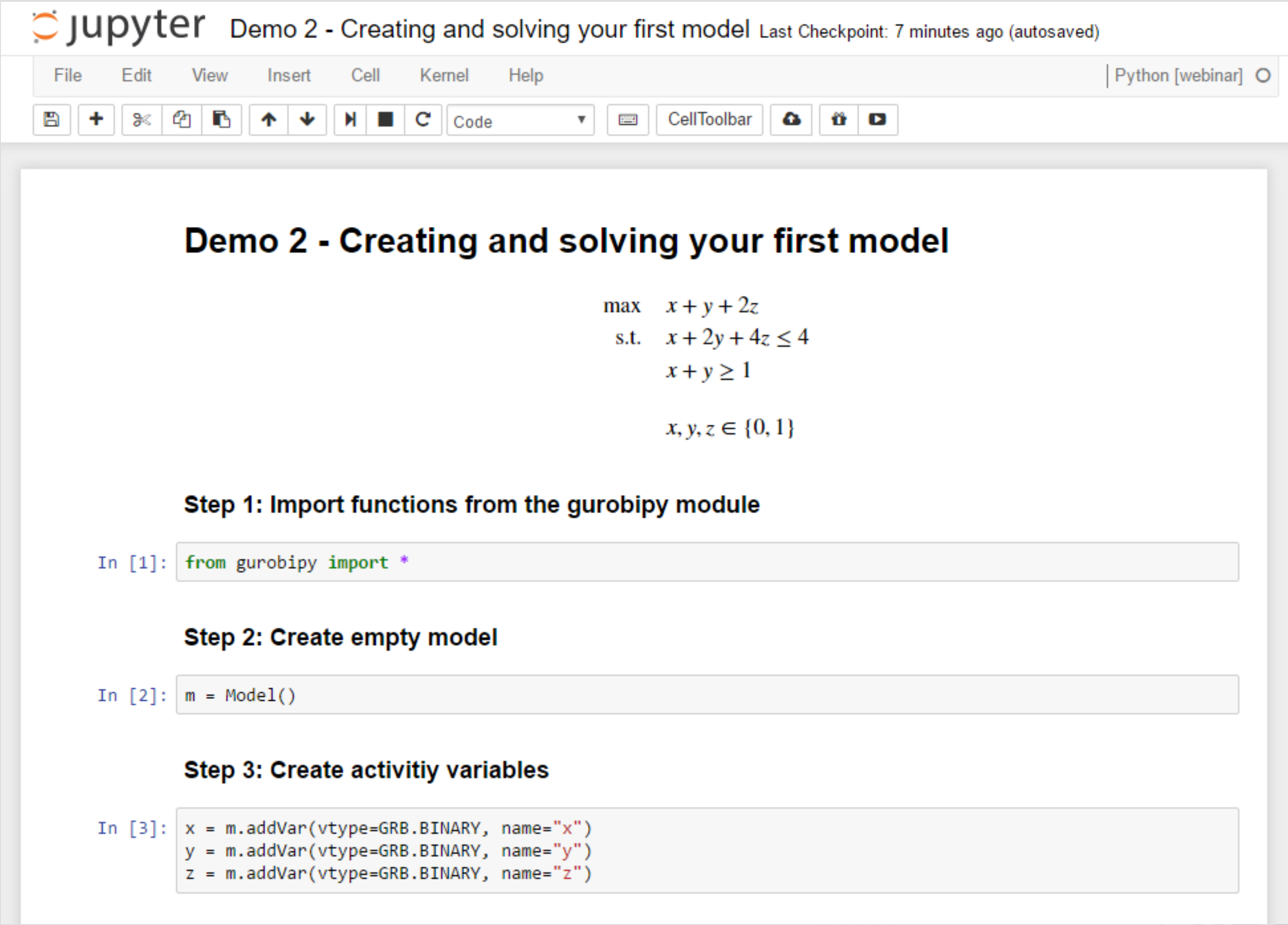
Creating and Solving Your First Model #2

- Open a new Jupyter Notebook
- Follow the Best Practices
  - Create activity variables
  - Set objective function
  - Create linear expressions and use them to create constraints
  - Call optimize()
- Print out results

This model is the mip1 example that you can find for all APIs in the **`examples`** directory of the Gurobi installation.

```
# Create empty Model
m = Model()

# Add variables
x = m.addVar(vtype=GRB.BINARY, name="x")
y = m.addVar(vtype=GRB.BINARY, name="y")
z = m.addVar(vtype=GRB.BINARY, name="z")

# Set objective function
m.setObjective(x + y + 2*z,
GRB.MAXIMIZE)

# Add constraints
c1 = m.addConstr(x + 2*y + 4*z <= 4)
c2 = m.addConstr(x + y >= 1)

# Solve model
m.optimize()
```

Live Demo: Creating and Solving Your First Model



From a mathematical model to a Python model

LP model

- A *linear program* (LP) is an optimization problem of the form

$$
\begin{array}{lll}
\text{minimize} & \sum_{j \in J} c_j \cdot x_j & \\
\text{subject to} & \sum_{j \in J} a_{ij} \cdot x_j = b_i & \forall i \in I \\
 & l_j \le x_j \le u_j & \forall j \in J
\end{array}
$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Decision variables

$$\text{minimize} \quad \sum_{j \in J} c_j \cdot \boxed{x_j}$$

$$\text{subject to} \quad \sum_{j \in J} a_{ij} \cdot \boxed{x_j} = b_i \qquad \forall i \in I$$

$$l_j \le \boxed{x_j} \le u_j \qquad \forall j \in J$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Objective function

$$
\begin{aligned}
\text{minimize} \quad & \boxed{\sum_{j \in J} c_j \cdot x_j} \\
\text{subject to} \quad & \sum_{j \in J} a_{ij} \cdot x_j = b_i \qquad \forall i \in I \\
& l_j \le x_j \le u_j \qquad \forall j \in J
\end{aligned}
$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Constraints

$$\text{minimize} \quad \sum_{j \in J} c_j \cdot x_j$$

$$\text{subject to} \quad \sum_{j \in J} a_{ij} \cdot x_j = b_i \qquad \forall i \in I$$

$$l_j \le x_j \le u_j \qquad \forall j \in J$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Data coefficients

$$
\begin{array}{lll}
\text{minimize} & \sum_{j \in J} \boxed{c_j} \cdot x_j & \\
\text{subject to} & \sum_{j \in J} \boxed{a_{ij}} \cdot x_j = \boxed{b_i} & \forall i \in I \\
& \boxed{l_j} \leq x_j \leq \boxed{u_j} & \forall j \in J
\end{array}
$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Index sets

$$
\begin{array}{lll}
\text{minimize} & \sum_{j \in J} c_j \cdot x_j & \\
\text{subject to} & \sum_{j \in J} a_{ij} \cdot x_j = b_i & \forall i \in I \\
 & l_j \leq x_j \leq u_j & \forall j \in J
\end{array}
$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Subscripts

$$\begin{aligned} \text{minimize} \quad & \sum_{j \in J} c_j \cdot x_j \\ \text{subject to} \quad & \sum_{j \in J} a_{ij} \cdot x_j = b_i \quad && \forall i \in I \\ & l_j \le x_j \le u_j && \forall j \in J \end{aligned}$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Arithmetic operators

$$
\begin{array}{lll}
\text{minimize} & \sum_{j \in J} c_j \cdot x_j & \\
\text{subject to} & \sum_{j \in J} a_{ij} \cdot x_j = b_i & \forall i \in I \\
 & l_j \leq x_j \leq u_j & \forall j \in J
\end{array}
$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Constraint operators

$$\text{minimize} \quad \sum_{j \in J} c_j \cdot x_j$$

$$\text{subject to} \quad \sum_{j \in J} a_{ij} \cdot x_j \boxed{=} b_i \qquad \forall i \in I$$

$$l_j \boxed{\leq} x_j \boxed{\leq} u_j \qquad \forall j \in J$$

LP model

- A *linear program* (LP) is an optimization problem of the form

For all operators

$$
\begin{array}{lll}
\text{minimize} & \sum_{j \in J} c_j \cdot x_j & \\
\text{subject to} & \sum_{j \in J} a_{ij} \cdot x_j = b_i & \forall i \in I \\
 & l_j \leq x_j \leq u_j & \forall j \in J
\end{array}
$$

LP model

- A *linear program* (LP) is an optimization problem of the form

Aggregate sum operators

$$
\begin{aligned}
\text{minimize} \quad & \boxed{\sum_{j \in J}} c_j \cdot x_j \\
\text{subject to} \quad & \boxed{\sum_{j \in J}} a_{ij} \cdot x_j = b_i \qquad \forall i \in I \\
& l_j \le x_j \le u_j \qquad \forall j \in J
\end{aligned}
$$

General optimization modeling constructs

- Decision variables
- Objective function
- Constraints

- Built with:
  - Coefficients
  - Indices and subscripts
  - Operators
    - Basic arithmetic (+, -, ×, ÷)
    - Constraint (≤, =, ≥)
    - For all
    - Aggregate sum

Enhancements to Gurobi Python interface

- High-level optimization modeling constructs embedded in Python API
  - Improved syntax (operator overloading)
  - Aggregate sum operator (`quicksum`)
  - Convenient data initialization (`multidict`)
  - Functionality for efficiently working with sparse data (`tuplelist`)

- Design goals:
  - Bring "feel" of a modeling language to the Python interface
  - Allow for code that is easy to write and maintain
  - Maintain unified design across all of our interfaces
  - Remain lightweight and efficient compared to solver alone

Ex:

$$x_i + y_i \leq 5, \forall i \in I \quad \Leftrightarrow$$

```
m.addConstrs(x[i] + y[i] <= 5
             for i in I)
```

- Python already provides much of what we need for representing data, indices and subscripts
  - Lists, tuples, dictionaries, loops, generator expressions, …

Python list comprehension

- List comprehension is compact way to create lists
  - ```
    sqrd = [i*i for i in range(5)]
    print sqrd  # displays [0, 1, 4, 9, 16]
    ```
- Can be used to create subsequences that satisfy certain conditions (ex: filtering a list)
  - ```
    bigsqrd = [i*i for i in range(5) if i*i >= 5]
    print bigsqrd  # displays [9, 16]
    ```
- Can be used with multiple for loops (ex: all combinations)
  - ```
    prod = [i*j for i in range(3) for j in range(4)]
    print prod  # displays [0, 0, 0, 0, 0, 1, 2, 3, 0, 2, 4, 6]
    ```
- Generator expression is similar, but no brackets (ex: argument to aggregate sum)
  - ```
    sumsqrd = sum(i*i for i in range(5))
    print sumsqrd  # displays 30
    ```
- “Feels” like algebraic notation

Sums for objective and constraints

Simple

- `sum()` method for a `tupledict` of `Var` objects

```
x = m.addVars(10,
         vtype=GRB.BINARY)

m.addConstr(x.sum() <= 1)
```

Powerful

- `sum()` function
  - Argument: a list or generator expression
  - Gurobi provides `quicksum()`, which is faster for large expressions of Var objects

```
x = m.addVars(10,
         vtype=GRB.BINARY)

m.addConstr(
    sum(x[i] for i in range(10))
    <= 1)
```

Iterating in Python

GUROBI OPTIMIZATION

- Loops
  - Iterate over collections of elements (list, dictionary, …)

```
for c in cities:
  print c # must indent all statements in loop
```

- List comprehension
  - Efficiently build lists via notation resembling mathematical sets

```
penaltyarcs = [a for a in arcs if cost[a] > 1000]
```

- Generator expressions
  - Similar syntax to list comprehension, used for function arguments

```
obj = quicksum(cost[a]*x[a] for a in arcs)
```

For-all loops in optimization models

Explicit

```
for i in I:
  m.addConstr(
    quicksum(a[i,j]*x[i,j]
             for j in J)
    <= 5)
```

Implicit

```
m.addConstrs(x.prod(a,i,'*')
             <= 5 for i in I)
```

$$\sum_{j \in J} a_{ij} x_{ij} \leq 5 \;\; \forall i \in I$$

Exercise #2 – Putting it all together

- Download file at http://files.gurobi.com/training/knapsack.zip and unzip `knapsack.py`
- Fill in the necessary sections to solve the following model:

maximize	$p_0 x_0 + \ldots + p_6 x_6$
subject to	$w_0 x_0 + \ldots + w_6 x_6 \leq c$
	$x_0, \ldots, x_6$ binary

- Note the data coefficients (p, w, c) have already been provided for you
- Run the program

- Notes/Hints:
  - Optimal value = 15; solution is x_0=1, x_3=1
  - Make sure to inspect the exported model `knapsack.lp` to verify model is correct
  - Use the documentation or `help()` if you get stuck

Console

```
$ gurobi.sh knapsack.py
Optimize a model with 1 rows, 7 columns and 7 nonzeros
Coefficient statistics:
  Matrix range    [2e+00, 9e+00]
  Objective range [3e+00, 9e+00]
  Bounds range    [1e+00, 1e+00]
  RHS range       [9e+00, 9e+00]
...
Explored 0 nodes (1 simplex iterations) in 0.00 seconds
Thread count was 4 (of 4 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective 1.500000000000e+01, best bound 1.500000000000e+01, gap 0.0%

    Variable            X
-------------------------
         x0             1
         x3             1
```

knapsack.py

```
from gurobipy import *

# define data coefficients
n = 7
p = [6, 5, 8, 9, 6, 7, 3]
w = [2, 3, 6, 7, 5, 9, 4]
c = 9

# create empty model
m = Model()

# add decision variables
x = m.addVars(n, vtype=GRB.BINARY, name='x')
```

knapsack.py

```
# set objective function
m.setObjective(x.prod(p), GRB.MAXIMIZE)

# add constraint
m.addConstr(x.prod(w)) <= c, name='knapsack')

# solve model
m.optimize()

# display solution
if m.SolCount > 0:
  m.printAttr('X')

# export model
m.write('knapsack.lp')
```

knapsack.lp

```
Maximize
  6 x0 + 5 x1 + 8 x2 + 9 x3 + 6 x4 + 7 x5 + 3 x6
Subject To
 knapsack: 2 x0 + 3 x1 + 6 x2 + 7 x3 + 5 x4 + 9 x5 + 4 x6 <= 9
Bounds
Binaries
 x0 x1 x2 x3 x4 x5 x6
End
```

What makes a model difficult?

Model size

- Models typically become large via copies
  - Ex: regions, products, time, …
- Reducing model size is an art
  - What should be modeled?
  - What should be approximated?
- Some constraints may be treated as “lazy” (pulled into model only when violated)
- Gurobi is parallel by default
  - Parallel MIP consumes memory
- Solver considerations:
  - Have enough physical memory (RAM) to load and solve model in memory
  - Use 64-bits
  - Try compute server or cloud

Presolve is your friend

- Collection of presolve reductions applied before algorithms
  - Reduces problem size
  - Tightens formulation

- Presolve is very effective and finds the obvious reductions
  - Users do not need to apply as many reductions as possible

- Limits to what presolve can do
  - Can’t find reductions that aren’t actually implied by the model
  - Users have better understanding of underlying problem being modeled

Frequency – A series of related models

- Models may not be so easy when there are many to solve

- Warm starts can often reduce solve times
  - Automatic
    - Modify a model in memory rather than create a new model
  - Manual
    - LP: basis and primal/dual starts
    - MIP: start vectors

- Sometimes warm starts hurt more than they help
  - Try solving from scratch via concurrent

Modifying a model

- Change coefficients
  - Objective
  - RHS
  - Matrix
  - Bounds
- Change variable types: continuous, integer, etc.
- Add/delete variables or constraints

- For small changes, modifying a model is more efficient than creating a new model
  - Reuse existing model data
  - Automatically use prior solution as warm-start for new model if possible
    - Some changes will force solver to discard LP basis

Example – Modifying a model

```
model = read('usa13509.mps')
model.optimize()

        Solved in 7940 iterations and 0.15 seconds
        Optimal objective  1.959148400e+07

x105 = model.getVarByName('x105')
x105.LB = 0.6
model.optimize()

        Solved in 3 iterations and 0.01 seconds
        Optimal objective  1.959149680e+07

model.reset()
model.optimize()

        Solved in 7931 iterations and 0.14 seconds
        Optimal objective  1.959149680e+07
```

Integer variables

- In most cases, integer variables make a model more difficult
- General integer variables tend to be more difficult than binary (0-1)
- Things to consider:
  - Which general integers are necessary?
  - Can some variables be approximated?

Quadratic expressions

- Quadratic expressions are much more complex than linear
  - Especially for constraints: quadratic constraints require the barrier method

- Quadratic is essential for some applications
  - Ex: financial risk, engineering

- Quadratic constraints should *never* be used for logical expressions
  - Ex: $x = 0$ or $y = 0$ should *not* be modeled by $x \cdot y = 0$
  - More about logical expressions later

General interface guidance

- All interfaces are lightweight and efficient
  - Use your programming needs to pick an interface

- Python is easiest Gurobi interface to get started with
  - Nothing additional to setup and configure
  - Interactive and no compiling necessary
  - Easy to write because structure is less rigid

- If you are using a solver-independent modeling system, enabling Gurobi is easy
  - Ex: In AMPL model file, add

```
option solver gurobi_ampl;
option gurobi_options 'mipfocus 1';
```

- Migrating from another solver or proprietary modeling language should be easier than you think
  - Visit https://www.gurobi.com/resources/switching-to-gurobi/switching-overview for more guidelines

Programming pitfalls

Gurobi environments

- Parameters are set on an environment
- Models are built from an environment
- Multiple models can be built from the same parent environment
  - Each model gets their own copy
- Once a model is created, subsequent changes to parent environment not reflected in copy
- Use `Model.set()` function to make parameter changes for the copy
  - Ex: set time limit of 3600 seconds for parent environment using Java interface
    `model.set(GRB.DoubleParam.TimeLimit, 3600);`
  - Ex: set presolve level to 2 for model's environment using Java interface
    `model.set(GRBIntParam.Presolve, 2);`

Lazy updates

- Lazy updates make Gurobi interfaces efficient
  - Changes are made in batches
  - Building internal data structures is much more efficient if done in a single run

- Since Gurobi Optimizer 7.0, the `update()` function is called automatically!

- The update() function is still called behind the scenes – to reference new model elements
  - Typically: between creating variables and constraints

- For best performance, create variables, then create constraints
  - Avoid a loop that creates a few variables then adds a few constraints

Memory management

- C++ considerations:
  - Always pass by reference, not by value
  - Be careful about an object's lifecycle (ex: destructor is called when they go out of scope)
  - Delete pointers to objects when finished, or you'll have a memory leak
  - Gurobi creates some objects on the heap (ex: `GRBModel::addVars`)

- Java and .NET considerations:
  - Garbage collector typically does not free `GRBModel` and `GRBEnv` objects instantaneously
    - Call the `dispose()` methods to explicitly free them

- Python considerations:
  - Garbage collector typically does not free `Model` objects instantaneously
    - Use `del m` to explicitly free them
  - Default environment not created until first used
    - Released on demand with new `disposeDefaultEnv()` method

Ignoring optimization status

- Input:

```
import sys
from gurobipy import *

m = read(sys.argv[1])
m.optimize()
for v in m.getVars():
  print v.VarName, v.X
```

- Output – runtime exception!

```
Model is infeasible
Best objective -, best bound -, gap -
x0
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    print v.VarName, v.X
  File "var.pxi", line 76, in gurobipy.Var.__getattr__ (../../src/python/gurobipy.c:11798)
  File "var.pxi", line 142, in gurobipy.Var.getAttr (../../src/python/gurobipy.c:12609)
gurobipy.GurobiError: Unable to retrieve attribute 'X'
```

Managing solution status

- Multiple outcomes possible for optimization models: optimal, infeasible, unbounded, …

- Check the `Status` attribute to see the result of the optimization

```
if m.Status == GRB.OPTIMAL:
  for v in m.getVars():
    print v.VarName, v.X
```

- Use `SolCount` attribute to see whether any solutions were found

```
if m.SolCount > 0:
  for v in m.getVars():
    print v.VarName, v.X
```

Error handling

- Programming errors often lead to unexpected errors at runtime

- Easy to catch exceptions in OO interfaces:

```
try:
  m = read(sys.argv[1])
  m.optimize()
  for v in m.getVars():
    print v.VarName, v.X
except GurobiError as e:
  print 'Error:', e
```

- With C, test the return code for every call to the Gurobi API

- Don't be sloppy – always test for errors!
  - Many support requests could be avoided by testing for and reviewing error codes

Model debugging

Common error types

- Model logic errors – when a model is written incorrectly
  - Can lead to no answers (infeasibility), wrong answers or suboptimal answers
    - Suboptimal answers are most difficult to test
    - How do you know when constraints incorrectly eliminate a valid solution?
  - Must keep code simple to read and understand

- Data errors – solving with bogus input data
  - Typically result of user errors at runtime
    - Be a defensive programmer and handle corner cases
  - Often lead to infeasible models

- Developing models requires testing, testing and more testing!

Model files

LP format

- Easy to read and understand
- May truncate some digits
- Order is not preserved

- Best for debugging

MPS format

- Machine-readable
- Full precision
- Order is preserved

- Best for testing

LP format example

```
Maximize
  x + y + 2 z
Subject To
 c0: x + 2 y + 3 z <= 4
 c1: x + y >= 1
Bounds
Binaries
 x y z
End
```

Naming variables and constraints

- Set the `VarName` and `ConstrName` attributes to meaningful values
  - `flow_Atlanta_Dallas` is more useful than `x3615`

- Don’t reuse names for multiple constraints or variables
  - API doesn’t care about the `VarName` or `ConstrName` attributes
  - Create unique, descriptive names to help with debugging

MPS format example

- `m.write("mymodel.mps");`

- Now, you can use this model file for any kind of tests
  - Command-line:
    ```
    $ gurobi_cl [parameters] mymodel.mps
    ```
  - Interactive shell:
    ```
    > m = read("mymodel.mps")
    > m.optimize()
    ```

- MPS files are a great way to export models from other solvers too
  - Useful for performance comparisons
  - Visit https://www.gurobi.com/resources/switching-to-gurobi/exporting-mps-files-from-competing-solvers for detailed instructions

Diagnosing infeasibility

- Unfortunately, it is not usually easy to diagnose

- If we know of feasible solution, then easier job to find problem
  - Evaluate existing constraints using variable values to find violations

- Gurobi provides Irreducible Infeasible Subsystem (IIS) detection
  - Finds a minimal subset of the constraints that is infeasible
  - Primarily used as a debugging tool

- Gurobi also supports constraint relaxations (feasRelax)
  - Find a solution that minimizes constraint violations (total, sum of squares or count)
  - Used as a debugging tool, or in production settings

Thank you – Questions?