Python API Overview

This section documents the Gurobi Python interface. It begins with an overview of the global functions, which can be called without referencing any Python objects. It then discusses the different types of objects that are available in the interface, and the most important methods on those objects. Finally, it gives a comprehensive presentation of all of the available classes and methods.

If you are new to the Gurobi Optimizer, we suggest that you start with the Quick Start Guide or the Example Tour. These documents provide concrete examples of how to use the classes and methods described here.

Important note for AIX users: due to limited Python support on AIX, our AIX port does not include the Python interface.

Global Functions

The Gurobi shell contains a set of Global Functions that can be called without referring to any Gurobi objects. The most important of these functions is probably the read function, which allows you to read a model from a file. Other useful global functions are system, which allows you to issue shell commands from within the Gurobi shell, models, which gives you a list of the currently loaded models, and disposeDefaultEnv, which disposes of the default environment. Other global functions allow you to read, modify, or write Gurobi parameters (readParams, setParam, and writeParams).

Models

Most actions in the Gurobi Python interface are performed by calling methods on Gurobi objects. The most commonly used object is the Model. A model consists of a set of decision variables (objects of class Var or MVar), a linear or quadratic objective function on these variables (specified using Model.setObjective), and a set of constraints on these variables (objects of class Constr, QConstr, SOS, or GenConstr). Each variable has an associated lower bound, upper bound, and type (continuous, binary, etc.). Each linear or quadratic constraint has an associated sense (less-than-or-equal, greater-than-or-equal, or equal), and right-hand side value. Refer to this section for more information on variables, constraints, and objectives.

An optimization model may be specified all at once, by loading the model from a file (using the previously mentioned read function), or it may be built incrementally, by first constructing an empty object of class Model and then subsequently calling Model.addVar, Model.addVars, or Model.addMVar to add additional variables, and Model.addConstr, Model.addConstrs, Model.addLConstr, Model.addQConstr, Model.addSOS, or any of the Model.addGenConstrXxx methods to add additional constraints.

Linear constraints are specified by building linear expressions (objects of class LinExpr or MLinExpr), and then specifying relationships between these expressions (for example, requiring that one expression be equal to another). Quadratic constraints are built in a similar fashion, but using quadratic expressions (objects of class QuadExpr or MQuadExpr) instead. General constraints are built using a set of dedicated methods, or a set of general constraint helper functions plus overloaded operators.

Models are dynamic entities; you can always add or remove variables or constraints.

We often refer to the *class* of an optimization model. A model with a linear objective function, linear constraints, and continuous variables is a *Linear Program (LP)*. If the objective is quadratic, the model is a *Quadratic Program (QP)*. If any of the constraints are quadratic, the model is a *Quadratically-Constrained Program (QCP)*. We will sometimes refer to a few special cases of QCP: QCPs with convex constraints, QCPs with non-convex constraints, *bilinear programs*, and *Second-Order Cone Programs (SOCP)*. If the model contains any integer variables, semi-continuous variables, semi-integer variables, Special Ordered Set (SOS) constraints, or general constraints, the model is a *Mixed Integer Program (MIP)*. We'll also sometimes discuss special cases of MIP, including *Mixed Integer Linear Programs (MILP)*, *Mixed Integer Quadratic Programs (MIQP)*, *Mixed Integer Quadratically-Constrained Programs (MIQCP)*, and *Mixed Integer Second-Order Cone Programs (MISOCP)*. The Gurobi Optimizer handles all of these model classes.

Environments

Environments play a much smaller role in the Gurobi Python interface than they do in our other language APIs, mainly because the Python interface has a default environment. Unless you explicitly pass your own environment to routines that require an environment, the default environment will be used.

The main situation where you may want to create your own environment is when you want precise control over when the resources associated with an environment (specifically, a licensing token or a Compute Server) are released. If you use your own environment to create models (using read or the Model constructor), then the resources associated with the environment will be released as soon your program no longer references your environment or any models created with that environment.

Note that you can manually remove the reference to the default environment, thus making it available for garbage collection, by calling disposeDefaultEnv. After calling this, and after all models built within the default environment are garbage collected, the default environment will be garbage collected as well. A new default environment will be created automatically if you call a routine that needs one.

For more advanced use cases, you can use an empty environment to create an uninitialized environment and then, programmatically, set all required options for your specific requirements. For further details see the Environment section.

Solving a Model

Once you have built a model, you can call Model.optimize to compute a solution. By default, optimize will use the concurrent optimizer to solve LP models, to solve LP models, the barrier algorithm to solve QP models with convex objectives and QCP models with convex constraints, and the branch-and-cut algorithm otherwise. The solution is stored in a set of *attributes* of the model, which can be subsequently queried (we will return to this topic shortly).

The Gurobi algorithms keep careful track of the state of the model, so calls to Model.optimize will only perform further optimization if relevant data has changed since the model was last optimized. If you would like to discard previously computed solution information and restart the optimization from scratch without changing the model, you can call Model.reset.

After a MIP model has been solved, you can call Model.fixed to compute the associated *fixed* model. This model is identical to the original, except that the integer variables are fixed to their values in the MIP solution. If your model contains SOS constraints, some continuous variables that appear in these constraints may be fixed as well. In some applications, it can be useful to compute information on this fixed model (e.g., dual variables, sensitivity information, etc.), although you should be careful in how you interpret this information.

Multiple Solutions, Objectives, and Scenarios

By default, the Gurobi Optimizer assumes that your goal is to find one proven optimal solution to a single model with a single objective function. Gurobi provides the following features that allow you to relax these assumptions:

- Solution Pool: Allows you to find more solutions.
- Multiple Scenarios: Allows you to find solutions to multiple, related models.
- Multiple Objectives: Allows you to specify multiple objective functions and control the trade-off between them.

Infeasible Models

You have a few options if a model is found to be infeasible. You can try to diagnose the cause of the infeasibility, attempt to repair the infeasibility, or both. To obtain information that can be useful for diagnosing the cause of an infeasibility, call Model.computeIIS to compute an Irreducible Inconsistent Subsystem (IIS). This method can be used for both continuous and MIP models, but you should be aware that the MIP version can be quite expensive. This method populates a set of IIS attributes.

To attempt to repair an infeasibility, call Model.feasRelaxS or Model.feasRelax to compute a feasibility relaxation for the model. This relaxation allows you to find a solution that minimizes the magnitude of the constraint violation.

Querying and Modifying Attributes

Most of the information associated with a Gurobi model is stored in a set of attributes. Some attributes are associated with the variables of the model, some with the constraints of the model, and some with the model itself. To give a simple example, solving an optimization model causes the `x` variable attribute to be populated. Attributes such as `x` that are computed by the Gurobi optimizer cannot be modified directly by the user, while others, such as the variable lower bound (the `lb` attribute) can.

Attributes can be accessed in two ways in the Python interface. The first is to use the `getAttr()` and `setAttr()` methods, which are available on variables (Var.getAttr/ Var.setAttr), linear constraints (Constr.getAttr/ Constr.setAttr), quadratic constraints (QConstr.getAttr/ QConstr.setAttr), SOSs (SOS.getAttr), general constraints (GenConstr.getAttr/ GenConstr.setAttr), and models (Model.getAttr/ Model.setAttr). These are called with the attribute name as the first argument (e.g., `var.getAttr("x")` or `constr.setAttr("rhs", 0.0)`). The full list of available attributes can be found in the Attributes section of this manual.

Attributes can also be accessed more directly: you can follow an object name by a period, followed by the name of an attribute of that object. Note that upper/lower case is ignored when referring to attributes. Thus, `b = constr.rhs` is equivalent to `b = constr.getAttr("rhs")`, and `constr.rhs = 0.0` is equivalent to `constr.setAttr("rhs", 0.0)`.

Additional Model Modification Information

Most modifications to an existing model are done through the attribute interface (e.g., changes to variable bounds, constraint right-hand sides, etc.). The main exceptions are modifications to the constraint matrix and to the objective function.

The constraint matrix can be modified in a few ways. The first is to call the Model.chgCoeff method. This method can be used to modify the value of an existing non-zero, to set an existing non-zero to zero, or to create a new non-zero. The constraint matrix is also modified when you remove a variable or constraint from the model (through the Model.remove method). The non-zero values associated with the deleted constraint or variable are removed along with the constraint or variable itself.

The model objective function can also be modified in a few ways. The easiest is to build an expression that captures the objective function (a LinExpr, MLinExpr, QuadExpr, or MQuadExpr object), and then pass that expression to method Model.setObjective. If you wish to modify the objective, you can simply call `setObjective` again with a new `LinExpr` or `QuadExpr` object.

For linear objective functions, an alternative to `setObjective` is to use the `Obj` variable attribute to modify individual linear objective coefficients.

If your variables have piecewise-linear objectives, you can specify them using the Model.setPWLObj method. Call this method once for each relevant variable. The Gurobi simplex solver includes algorithmic support for convex piecewise-linear objective functions, so for continuous models you should see a substantial performance benefit from using this feature. To clear a previously specified piecewise-linear objective function, simply set the `Obj` attribute on the corresponding variable to 0.

Lazy Updates

One important item to note about model modification in the Gurobi optimizer is that it is performed in a *lazy* fashion, meaning that modifications don't affect the model immediately. Rather, they are queued and applied later. If your program simply creates a model and solves it, you will probably never notice this behavior. However, if you ask for information about the model before your modifications have been applied, the details of the lazy update approach may be relevant to you.

As we just noted, model modifications (bound changes, right-hand side changes, objective changes, etc.) are placed in a queue. These queued modifications can be applied to the model in three different ways. The first is by an explicit call to Model.update. The second is by a call to Model.optimize. The third is by a call to Model.write to write out the model. The first case gives you fine-grained control over when modifications are applied. The second and third make the assumption that you want all pending modifications to be applied before you optimize your model or write it to disk.

Why does the Gurobi interface behave in this manner? There are a few reasons. The first is that this approach makes it much easier to perform multiple modifications to a model, since the model remains unchanged between modifications. The second is that processing model modifications can be expensive, particularly in a Compute Server environment, where modifications require communication between machines. Thus, it is useful to have visibility into exactly when these modifications are applied. In general, if your program needs to make multiple modifications to the model, you should aim to make them in phases, where you make a set of modifications, then update, then make more modifications, then update again, etc. Updating after each individual modification can be extremely expensive.

If you forget to call update, your program won't crash. Your query will simply return the value of the requested data from the point of the last update. If the object you tried to query didn't exist then, you'll get a `NOT_IN_MODEL` exception instead.

The semantics of lazy updates have changed since earlier Gurobi versions. While the vast majority of programs are unaffected by this change, you can use the UpdateMode parameter to revert to the earlier behavior if you run into an issue.

Managing Parameters

The Gurobi optimizer provides a set of parameters that allow you to control many of the details of the optimization process. Factors like feasibility and optimality tolerances, choices of algorithms, strategies for exploring the MIP search tree, etc., can be controlled by modifying Gurobi parameters before beginning the optimization. Parameters are set using method Model.setParam. Current values may also be retrieved with Model.getParamInfo. You can also access parameters more directly through the `Model.Params` class. To set the MIPGap parameter to 0.0 for model `m`, for example, you can do either `m.setParam('MIPGap', 0)` or `m.Params.MIPGap = 0`.

You can read a set of parameter settings from a file using Model.read, or write the set of changed parameters using Model.write.

We also include an automated parameter tuning tool that explores many different sets of parameter changes in order to find a set that improves performance. You can call Model.tune to invoke the tuning tool on a model. Refer to the parameter tuning tool section for more information.

One thing we should note is that changing a parameter for one model has no effect on the parameter value for other models. Use the global setParam method to set a parameter for all loaded models.

The full list of Gurobi parameters can be found in the Parameters section.

Monitoring Progress - Logging and Callbacks

Progress of the optimization can be monitored through Gurobi logging. By default, Gurobi will send output to the screen. A few simple controls are available for modifying the default logging behavior. You can set the LogFile parameter if you wish to also direct the Gurobi log to a file. The frequency of logging output can be controlled with the DisplayInterval parameter, and logging can be turned off entirely with the OutputFlag parameter.

Log output is also sent to a Python logger named `gurobipy`, at level `INFO`. You can use the Python `logging` module to connect to this log.

More detailed progress monitoring can be done through a callback function. If you pass a function taking two arguments, `model` and `where`, to Model.optimize, your function will be called periodically from within the optimization. Your callback can then call Model.cbGet to retrieve additional information on the state of the optimization. You can refer to the Callback class for additional information.

Modifying Solver Behavior - Callbacks

Callbacks can also be used to modify the behavior of the Gurobi optimizer. The simplest control callback is Model.terminate, which asks the optimizer to terminate at the earliest convenient point. Method Model.cbSetSolution allows you to inject a feasible solution (or partial solution) during the solution of a MIP model. Methods Model.cbCut and Model.cbLazy allow you to add *cutting planes* and *lazy constraints* during a MIP optimization, respectively. Method Model.cbStopOneMultiObj allows you to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

Batch Optimization

Gurobi Compute Server enables programs to offload optimization computations onto dedicated servers. The Gurobi Cluster Manager adds a number of additional capabilities on top of this. One important one, *batch optimization*, allows you to build an optimization model with your client program, submit it to a Compute Server cluster (through the Cluster Manager), and later check on the status of the model and retrieve its solution. You can use a Batch object to make it easier to work with batches. For details on batches, please refer to the Batch Optimization section.

Error Handling

All of the methods in the Gurobi Python library can throw an exception of type GurobiError. When an exception occurs, additional information on the error can be obtained by retrieving the `errno` or `message` members of the `GurobiError` object. A list of possible values for the `errno` field can be found in the Error Code section.

6.1 Global Functions

Gurobi global functions. These functions can be accessed from the main Gurobi shell prompt. In contrast to all other methods in the Gurobi Python interface, these functions do not require a Gurobi object to invoke them.

models()

```
models ( )
```

Print a list of loaded models.

Note that this function will only list models stored in global variables. Models stored in Python data structures (lists, dictionaries, etc.), or inside user classes aren't listed.

Example usage:

```
a = Model("a")
b = Model("b")
models()
```

disposeDefaultEnv()

```
disposeDefaultEnv ( )
```

Dispose of the default environment.

Calling this function releases the default environment created by the Gurobi Python module. This function is particularly useful in a long-running Python session (e.g., within a Jupyter notebook), where the Gurobi environment would otherwise continue to exist for the full duration of the session.

Note that models built with the default environment must be garbage collected before the default environment can be freed. You can force a model `m` to be garbage collected with the statement `del m`. If no references to the default environment remain, `disposeDefaultEnv` prints the message

```
Freed default Gurobi environment
```

to confirm it was able to dispose of the default environment.

Example usage:

```
disposeDefaultEnv()
```

multidict()

```
multidict ( data )
```

This function splits a single dictionary into multiple dictionaries. The input dictionary should map each key to a list of `n` values. The function returns a list of the shared keys as its first result, followed by the `n` individual Gurobi tuple dictionaries (stored as tupledict objects).

Arguments:

`data`: A Python dictionary. Each key should map to a list of values.

Return value:

A list, where the first member contains the shared key values, and the following members contain the dictionaries that result from splitting the value lists from the input dictionary.

Example usage:

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

paramHelp()

`paramHelp ( paramname )`

Obtain help about a Gurobi parameter.

Arguments:

`paramname`: String containing the name of parameter that you would like help with. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed. Note that case is ignored.

Example usage:

```
paramHelp("Cuts")
paramHelp("Heu*")
paramHelp("*cuts")
```

quicksum()

`quicksum ( data )`

A version of the Python `sum` function that is much more efficient for building large Gurobi expressions (LinExpr or QuadExpr objects). The function takes a list of terms as its argument.

Note that while `quicksum` is much faster than `sum`, it isn't the fastest approach for building a large expression. Use addTerms or the LinExpr() constructor if you want the quickest possible expression construction.

Arguments:

`data`: List of terms to add. The terms can be constants, Var objects, LinExpr objects, or QuadExpr objects.

Return value:

An expression that represents the sum of the terms in the input list.

Example usage:

```
expr = quicksum([2*x, 3*y+1, 4*z*z])
expr = quicksum(model.getVars())
```

read()

```
read  ( filename, env=defaultEnv )
```

Read a model from a file.

Arguments:

`filename`: Name of file containing model. Note that the type of the file is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, `.rlp`, `.ilp`, or `.opb`. The files can be compressed, so additional suffixes of `.gz`, `.bz2`, `.zip`, or `.7z` are accepted. The file name may contain `*` or `?` wildcards. No file is read when no wildcard match is found. If more than one match is found, this routine will attempt to read the first matching file.

`env`: Environment in which to create the model. Creating your environment (using the Env constructor) gives you more control over Gurobi licensing, but it can make your program more complex. Use the default environment unless you know that you need to control your own environments.

Return value:

Model object containing the model that was read from the input file.

Example usage:

```
m = read("afiro.mps")
m.optimize()
```

readParams()

```
readParams  ( filename )
```

Read a set of parameter settings from a file. The file name must end in `.prm`, and the file must be in PRM format.

Arguments:

`filename`: Name of file containing parameter settings.

Example usage:

```
readParams("params.prm")
```

resetParams()

```
resetParams  ( )
```

Reset the values of all parameters to their default values. Note that existing models that are stored inside Python data structures (lists, dictionaries, etc.), or inside user classes aren't affected.

Example usage:

```
resetParams()
```

setParam()

```
setParam  ( paramname, newvalue )
```

Set the value of a parameter to a new value. Note that existing models that are stored inside Python data structures (lists, dictionaries, etc.), or inside user classes aren't affected.

Arguments:

paramname: String containing the name of parameter that you would like to modify. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and none are modified. Note that case is ignored.

newvalue: Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

Example usage:

```
setParam("Cuts", 2)
setParam("Heu*", 0.5)
setParam("*Interval", 10)
```

system()

```
system ( command )
```

Issue a system command.

Arguments:

command: A string containing the desired system command.

Example usage:

```
system("ls")
system("rm junk")
```

writeParams()

```
writeParams ( filename )
```

Write all modified parameters to a file. The file is written in PRM format.

Example usage:

```
setParam("Heu*", 0.5)
writeParams("params.prm")  # file will contain changed parameter
system("cat params.prm")
```

6.2 Model

Gurobi model object. Commonly used methods on the model object in the Gurobi shell include optimize (optimizes the model), printStats (prints statistics about the model), printAttr (prints the values of an attribute), and write (writes information about the model to a file). Commonly used methods when building a model include addVar (adds a new variable), addVars (adds multiple new variables), addMVar (adds an a NumPy ndarray of Gurobi variables), addConstr (adds a new constraint), and addConstrs (adds multiple new constraints).

Model()

```
Model ( name="", env=defaultEnv )
```

Model constructor.

Arguments:

`name`: Name of new model. Note that **`name`** will be stored as an ASCII string. Thus, a name like **`'A→B'`** will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

`env`: Environment in which to create the model. Creating your environment (using the Env constructor) gives you more control over Gurobi licensing, but it can make your program more complex. Use the default environment unless you know that you need to control your own environments.

Return value:

New model object. Model initially contains no variables or constraints.

Example usage:

```
m = Model("NewModel")
x0 = m.addVar()

env = Env("my.log")
m2 = Model("NewModel2", env)
```

Model.addConstr()

```
addConstr ( lhs, sense=None, rhs=None, name="" )
```

Add a constraint to a model.

Note that this method also accepts a TempConstr as its first argument (with the constraint name as its second argument). This allows you to use operator overloading to create a variety of different constraint types. See TempConstr for more information.

Arguments:

`lhs`: Left-hand side for the new constraint. Can be a constant, a Var, a LinExpr, a QuadExpr, or a TempConstr.

`sense`: Sense for the new constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

`rhs`: Right-hand side for the new constraint. Can be a constant, a Var, a LinExpr, or a QuadExpr.

name: Name for new constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New constraint object.

Example usage:

```
model.addConstr(x + 2*y, GRB.EQUAL, 3*z, "c0")
model.addConstr(x + y <= 2.0, "c1")
model.addConstr(x*x + y*y <= 4.0, "qc0")
model.addConstr(x + y + z == [1, 2], "rgc0")
model.addConstr(z == and_(x, y, w), "gc0")
model.addConstr(z == min_(x, y), "gc1")
model.addConstr((w == 1) >> (x + y <= 1), "ic0")
```

Warning: A constraint can only have a single comparison operator.
While `1 <= x + y <= 2` or `1 <= x[i] + y[i] <= 2 for i in range(3)`
may look like valid constraints, our Python API won't interpret them as they were intended, which will almost
certainly result in unexpected behavior.

Model.addConstrs()

addConstrs (generator, name="")

Add multiple constraints to a model using a Python generator expression. Returns a Gurobi tupledict that contains the newly created constraints, indexed by the values generated by the generator expression.

The first argument to `addConstrs` is a Python generator expression, a special feature of the Python language that allows you to iterate over a Python expression. In this case, the Python expression will be a Gurobi constraint and the generator expression provides values to plug into that constraint. A new Gurobi constraint is added to the model for each iteration of the generator expression.

To give an example, if `x` is a Gurobi variable, then

```
m.addConstr(x <= 1, name='c0')
```

would add a single linear constraint involving this variable. In contrast, if `x` is a list of Gurobi variables, then

```
m.addConstrs((x[i] <= 1 for i in range(4)), name='c')
```

would add four constraints to the model. The entire first argument is a generator expression, where the indexing is controlled by the statement `for i in range(4)`, The first constraint that results from this expression would be named `c[0]`, and would involve variable `x[0]`. The second would be named `c[1]`, and would involve variable `x[1]`.

Generator expressions can be much more complex than this. They can involve multiple variables and conditional tests. For example, you could do:

```
m.addConstrs((x[i,j] == 0 for i in range(4)
                          for j in range(4)
                          if i != j), name='c')
```

One restriction that `addConstrs` places on the generator expression is that each variable must always take a scalar value (`int`, `float`, `string`, ...). Thus, `for i in [1, 2.0, 'a', 'bc']` is fine, but `for i in [(1, 2), [1, 2, 3]]` isn't.

This method can be used to add linear constraints, quadratic constraints, or general constraints to the model. Refer to the TempConstr documentation for more information on all of the different constraint types that can be added.

Note that if you supply a name argument, the generator expression must be enclosed in parenthesis. This requirement comes from the Python language interpreter.

Arguments:

`generator`: A generator expression, where each iteration produces a constraint.

`name`: Name pattern for new constraints. The given name will be subscripted by the index of the generator expression, so if the index is an integer, `c` would become `c[0]`, `c[1]`, etc. Note that the generated names will be stored as ASCII strings, so you should avoid using names that contain non-ASCII characters. In addition, names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

A dictionary of Constr objects, indexed by the values specified by the generator expression.

Example usage:

```
model.addConstrs(x.sum(i, '*') <= capacity[i] for i in range(5))
model.addConstrs(x[i] + x[j] <= 1 for i in range(5) for j in range(5))
model.addConstrs(x[i]*x[i] + y[i]*y[i] <= 1 for i in range(5))
model.addConstrs(x.sum(i, '*') == [0, 2] for i in [1, 2, 4])
model.addConstrs(z[i] == max_(x[i], y[i]) for i in range(5))
model.addConstrs((x[i] == 1) >> (y[i] + z[i] <= 5) for i in range(5))
```

Warning: A constraint can only have a single comparison operator.
While `1 <= x + y <= 2` or `1 <= x[i] + y[i] <= 2 for i in range(3)`
may look like valid constraints, our Python API won't interpret them as they were intended, which will almost
certainly result in unexpected behavior.

Model.addGenConstrXxx()

Each of the functions described below adds a new *general constraint* to a model.

Mathematical programming has traditionally defined a set of fundamental constraint types: variable bound constraints, linear constraints, quadratic constraints, integrality constraints, and SOS constraints. These are typically treated directly by the underlying solver (although not always), and are fundamental to the overall algorithm.

Gurobi accepts a number of additional constraint types, which we collectively refer to as *general (function) constraints*. These are typically *not* treated directly by the solver. Rather, they are transformed by presolve into constraints (and variables) chosen from among the fundamental types listed above. In some cases, the resulting constraint or constraints are mathematically equivalent to the original; in others, they are approximations. If such constraints appear in your model, but if you prefer to reformulate them yourself using fundamental constraint types instead, you can certainly do so. However, note that Gurobi can sometimes exploit information contained in the other constraints in the model to build a more efficient formulation than what you might create.

The additional constraint types that fall under this *general constraint* umbrella are:

- addGenConstrMax: $y = max(x_1, x_2, ..., c)$
- addGenConstrMin: $y = min(x_1, x_2, ..., c)$
- addGenConstrAbs: $y = |x|$
- addGenConstrAnd: $y = x_1 \wedge x_2 \wedge x_3 ...$
- addGenConstrOr: $y = x_1 \vee x_2 \vee x_3 ...$
- addGenConstrIndicator: $y = 1 \rightarrow a'x \leq b$ (an indicator constraint)
- addGenConstrPWL: $y = pwl(x)$ (a piecewise-linear function, specified using breakpoints)
- addGenConstrPoly: $y = p_0 x^d + p_1 x^{d-1} + ... + p_{d-1} x + p_d$
- addGenConstrExp: $y = e^x$
- addGenConstrExpA: $y = a^x$
- addGenConstrLog: $y = log_e(x)$
- addGenConstrLogA: $y = log_a(x)$
- addGenConstrPow: $y = x^a$
- addGenConstrSin: $y = sin(x)$
- addGenConstrCos: $y = cos(x)$
- addGenConstrTan: $y = tan(x)$

You can also add several types of general constraints through addConstr or addConstrs, using overloaded operators and a few general constraint helper functions. The descriptions below will make note of these equivalent, more concise alternatives.

Please refer to this section for additional details on general constraints.

Model.addGenConstrMax()

```
addGenConstrMax ( resvar, vars, constant=None, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_MAX` to a model.

A MAX constraint $r = \max\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the maximum of the operand variables $x_1, \ldots, x_n$ and the constant c.

You can also add a MAX constraint using the max_ function.

Arguments:

resvar (Var): The variable whose value will be equal to the max of the other variables.

vars (list of Var): The variables over which the max will be taken. Note that this list may also contain constants (type int, long, or float).

constant (float, optional): An additional operand that allows you to include a constant among the arguments of the max operation.

name (string, optional): Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = max(x1, x3, x4, 2.0)
model.addGenConstrMax(x5, [x1, x3, x4], 2.0, "maxconstr")

# alternative form
model.addGenConstrMax(x5, [x1, x3, x4, 2.0], name="maxconstr")

# overloaded forms
model.addConstr(x5 == max_([x1, x3, x4, 2.0]), name="maxconstr")
model.addConstr(x5 == max_(x1, x3, x4, 2.0), name="maxconstr")
```

Model.addGenConstrMin()

```
addGenConstrMin ( resvar, vars, constant=None, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_MIN` to a model.

A MIN constraint $r = \min\{x_1, \ldots, x_n, c\}$ states that the resultant variable r should be equal to the minimum of the operand variables $x_1, \ldots, x_n$ and the constant c.

You can also add a MIN constraint using the min_ function.

Arguments:

resvar (Var): The variable whose value will be equal to the min of the other variables.

vars (list of Var): The variables over which the min will be taken. Note that this list may also contain constants (type int, long, or float).

constant (float, optional): An additional operand that allows you to include a constant among the arguments of the min operation.

name (string, optional): Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = min(x1, x3, x4, 2.0)
model.addGenConstrMin(x5, [x1, x3, x4], 2.0, "minconstr")

# alternative form
model.addGenConstrMin(x5, [x1, x3, x4, 2.0], name="minconstr")

# overloaded forms
```

```
model.addConstr(x5 == min_([x1, x3, x4, 2.0]), name="minconstr")
model.addConstr(x5 == min_(x1, x3, x4, 2.0), name="minconstr")
```

Model.addGenConstrAbs()

`addGenConstrAbs ( resvar, argvar, name="" )`

Add a new general constraint of type `GRB.GENCONSTR_ABS` to a model.

An ABS constraint $r = \text{abs}\{x\}$ states that the resultant variable r should be equal to the absolute value of the argument variable x.

You can also add an ABS constraint using the abs_ function.

Arguments:

`resvar (Var)`: The variable whose value will be to equal the absolute value of the argument variable.

`argvar (Var)`: The variable for which the absolute value will be taken.

`name (string, optional)`: Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = abs(x1)
model.addGenConstrAbs(x5, x1, "absconstr")

# overloaded form
model.addConstr(x5 == abs_(x1), name="absconstr")
```

Model.addGenConstrAnd()

`addGenConstrAnd ( resvar, vars, name="" )`

Add a new general constraint of type `GRB.GENCONSTR_AND` to a model.

An AND constraint $r = \text{and}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if all of the operand variables $x_1, \ldots, x_n$ are equal to 1. If any of the operand variables is 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

You can also add an AND constraint using the and_ function.

Arguments:

`resvar (Var)`: The variable whose value will be equal to the AND concatenation of the other variables.

`vars (list of Var)`: The variables over which the AND concatenation will be taken.

`name (string, optional)`: Name for the new general constraint. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = and(x1, x3, x4)
model.addGenConstrAnd(x5, [x1, x3, x4], "andconstr")

# overloaded forms
model.addConstr(x5 == and_([x1, x3, x4]), "andconstr")
model.addConstr(x5 == and_(x1, x3, x4), "andconstr")
```

Model.addGenConstrOr()

```
addGenConstrOr ( resvar, vars, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_OR` to a model.

An OR constraint $r = \text{or}\{x_1, \ldots, x_n\}$ states that the binary resultant variable r should be 1 if and only if any of the operand variables $x_1, \ldots, x_n$ is equal to 1. If all operand variables are 0, then the resultant should be 0 as well.

Note that all variables participating in such a constraint will be forced to be binary, independent of how they were created.

You can also add an OR constraint using the or_ function.

Arguments:

`resvar (Var)`: The variable whose value will be equal to the OR concatenation of the other variables.

`vars (list of Var)`: The variables over which the OR concatenation will be taken.

`name (string, optional)`: Name for the new general constraint. Note that **`name`** will be stored as an ASCII string. Thus, a name like **`'A→B'`** will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x5 = or(x1, x3, x4)
model.addGenConstrOr(x5, [x1, x3, x4], "orconstr")

# overloaded forms
model.addConstr(x5 == or_([x1, x3, x4]), "orconstr")
model.addConstr(x5 == or_(x1, x3, x4), "orconstr")
```

Model.addGenConstrIndicator()

```
addGenConstrIndicator ( binvar, binval, lhs, sense=None, rhs=None, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_INDICATOR` to a model.

An INDICATOR constraint $z = f \rightarrow a^T x \leq b$ states that if the binary indicator variable z is equal to $f \in \{0, 1\}$, then the linear constraint $a^T x \leq b$ should hold. On the other hand, if $z = 1 - f$, the linear constraint may be violated. The sense of the linear constraint can also be specified to be $=$ or $\geq$.

Note that the indicator variable z of a constraint will be forced to be binary, independent of how it was created.

You can also add an INDICATOR constraint using a special overloaded syntax. See the examples below for details.

Arguments:

`binvar (Var)`: The binary indicator variable.

`binval (Boolean)`: The value for the binary indicator variable that would force the linear constraint to be satisfied.

`lhs (float, Var, LinExpr, or TempConstr)`: Left-hand side expression for the linear constraint triggered by the indicator. Can be a constant, a Var, or a LinExpr. Alternatively, a temporary constraint object can be used to define the linear constraint that is triggered by the indicator. The temporary constraint object is created using an overloaded comparison operator. See TempConstr for more information. In this case, the "sense" and "rhs" parameters must stay at their default values `None`.

`sense (char)`: Sense for the linear constraint. Options are `GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`.

`rhs (float)`: Right-hand side value for the linear constraint.

`name (string, optional)`: Name for the new general constraint. Note that **`name`** will be stored as an ASCII string. Thus, a name like **`'A→B'`** will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# x7 = 1 -> x1 + 2 x3 + x4 = 1
model.addGenConstrIndicator(x7, True, x1 + 2*x2 + x4, GRB.EQUAL, 1.0)

# alternative form
model.addGenConstrIndicator(x7, True, x1 + 2*x2 + x4 == 1.0)

# overloaded form
model.addConstr((x7 == 1) >> (x1 + 2*x2 + x4 == 1.0))
```

Model.addGenConstrPWL()

```
addGenConstrPWL  ( xvar, yvar, xpts, ypts, name="" )
```

Add a new general constraint of type `GRB.GENCONSTR_PWL` to a model.

A piecewise-linear (PWL) constraint states that the relationship $y = f(x)$ must hold between variables x and y, where f is a piecewise-linear function. The breakpoints for f are provided as arguments. Refer to the description of piecewise-linear objectives for details of how piecewise-linear functions are defined.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`xpts (list of float)`: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`ypts (list of float)`: The y values for the points that define the piecewise-linear function.

`name (string, optional)`: Name for the new general constraint.

Return value:

New general constraint.

Example usage:

```
gc = model.addGenConstrPWL(x, y, [0, 1, 2], [1.5, 0, 3], "myPWLConstr")
```

Model.addGenConstrPoly()

```
addGenConstrPoly ( xvar, yvar, p, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_POLY` to a model.

A polynomial function constraint states that the relationship $y = p_0x^d + p_1x^{d-1} + ... + p_{d-1}x + p_d$ should hold between variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`p`: The coefficients for the polynomial function (starting with the coefficient for the highest power).

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = 2 x^3 + 1.5 x^2 + 1
gc = model.addGenConstrPoly(x, y, [2, 1.5, 0, 1])
```

Model.addGenConstrExp()

```
addGenConstrExp ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_EXP` to a model.

A natural exponential function constraint states that the relationship $y = exp(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

yvar (Var): The y variable.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = exp(x)
gc = model.addGenConstrExp(x, y)
```

Model.addGenConstrExpA()

```
addGenConstrExpA ( xvar, yvar, a, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_EXPA` to a model.

An exponential function constraint states that the relationship $y = a^x$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.

yvar (Var): The y variable.

a (float): The base of the function, $a > 0$.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = 3^x
gc = model.addGenConstrExpA(x, y, 3.0, "expa")
```

Model.addGenConstrLog()

```
addGenConstrLog ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_LOG` to a model.

A natural logarithmic function constraint states that the relationship $y = log(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = ln(x)
gc = model.addGenConstrLog(x, y)
```

Model.addGenConstrLogA()

```
addGenConstrLogA ( xvar, yvar, a, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_LOGA` to a model.

A logarithmic function constraint states that the relationship $y = log_a(x)$ should hold for variables x and y, where $a > 0$ is the (constant) base.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`a (float)`: The base of the function, $a > 0$.

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = log10(x)
```

```
gc = model.addGenConstrLogA(x, y, 10.0, "log10", "FuncPieces=-1 FuncPieceError=1e-5")
```

Model.addGenConstrPow()

addGenConstrPow (xvar, yvar, a, name="", options="")

Add a new general constraint of type `GRB.GENCONSTR_POW` to a model.

A power function constraint states that the relationship $y = x^a$ should hold for variables x and y, where a is the (constant) exponent. The lower bound of variable x must be nonnegative, even if a is an integer.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`a (float)`: The exponent of the function, $a > 0$.

`name (string, optional)`: Name for the new general constraint.

`options (string, optional)`: A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = x^3.5
gc = model.addGenConstrLogA(x, y, 3.5, "gf", "FuncPieces=1000")
```

Model.addGenConstrSin()

addGenConstrSin (xvar, yvar, name="", options="")

Add a new general constraint of type `GRB.GENCONSTR_SIN` to a model.

A sine function constraint states that the relationship $y = sin(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`name (string, optional)`: Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = sin(x)
gc = model.addGenConstrSin(x, y)
```

Model.addGenConstrCos()

```
addGenConstrCos ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_COS` to a model.

A cosine function constraint states that the relationship $y = cos(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.

yvar (Var): The y variable.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = cos(x)
gc = model.addGenConstrCos(x, y)
```

Model.addGenConstrTan()

```
addGenConstrTan ( xvar, yvar, name="", options="" )
```

Add a new general constraint of type `GRB.GENCONSTR_TAN` to a model.

A tangent function constraint states that the relationship $y = tan(x)$ should hold for variables x and y.

A piecewise-linear approximation of the function is added to the model. The details of the approximation are controlled using the following four attributes (or using the parameters with the same names): FuncPieces, FuncPieceError, FuncPiecesLength, and FuncPieceRatio. For details, consult the General Constraint discussion.

Arguments:

xvar (Var): The x variable.

yvar (Var): The y variable.

name (string, optional): Name for the new general constraint.

options (string, optional): A string that can be used to set the attributes that control the piecewise-linear approximation of this function constraint. To assign a value to an attribute, follow the attribute name with an equal sign and the desired value (with no spaces). Assignments for different attributes should be separated by spaces (e.g. "FuncPieces=-1 FuncPieceError=0.001").

Return value:

New general constraint.

Example usage:

```
# y = tan(x)
gc = model.addGenConstrTan(x, y)
```

Model.addLConstr()

```
addLConstr ( lhs, sense=None, rhs=None, name="" )
```

Add a linear constraint to a model. This method is faster than addConstr() (as much as 50% faster for very sparse constraints), but can only be used to add linear constraints.

Note that this method also accepts a TempConstr as its first argument (with the name as its second argument). This allows you to use operator overloading to create constraints. See TempConstr for more information.

Arguments:

lhs: Left-hand side for the new constraint. Can be a constant, a Var, a LinExpr, or a TempConstr (while the TempConstr can only be of linear form).

sense: Sense for the new constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).

rhs: Right-hand side for the new constraint. Can be a constant, a Var, or a LinExpr.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New constraint object.

Example usage:

```
model.addLConstr(x + 2*y, GRB.EQUAL, 3*z, "c0")
model.addLConstr(x + y <= 2.0, "c1")
model.addLConstr(LinExpr([1.0,1.0], [x,y]), GRB.LESS_EQUAL, 1)
```

Model.addMConstrs()

```
addMConstrs ( A, x, sense, b, names="" )
```

Add a set of linear constraints to the model using matrix semantics. The added constraints are $Ax = b$ (except that the constraint sense is determined by the `sense` argument). The `A` argument must be a NumPy dense ndarray or a SciPy sparse matrix.

Note that you will typically use overloaded operators to build and add constraints using matrix semantics. The overloaded `@` operator can be used to build a linear matrix expression, which can then be used with an overloaded comparison operator to build a TempConstr object. This can then be passed to addConstr.

Arguments:

`A`: The constraint matrix - a NumPy 2-D dense ndarray or a SciPy sparse matrix.

`x`: Decision variables. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the size of the second dimension of `A`.

`sense`: Constraint senses, provided as a NumPy 1-D ndarray or as a single character. Valid values are $<$, $>$, or $=$. The length of the array must be equal the size of the first dimension of `A`. A character will be promoted to an ndarray of the appropriate length.

`b`: Right-hand side vector, stored as a NumPy 1-D ndarray. The length of the array must be equal the size of the first dimension of `A`.

`names`: Names for new constraints. The given name will be subscripted by the index of the constraint in the matrix.

Return value:

List of Constr objects.

Example usage:

```
A = np.full((5, 10), 1)
x = model.addMVar(10)
b = np.full(5, 1)

model.addMConstrs(A, x, '=', b)
```

Model.addMQConstr()

```
addMQConstr ( Q, c, sense, rhs, xQ_L=None, xQ_R=None, xc=None, name="" )
```

Add a quadratic constraint to the model using matrix semantics. The added constraint is $x_{Q_L}' Q x_{Q_R} + c' x_c = \mathrm{rhs}$ (except that the constraint sense is determined by the `sense` argument). The `Q` argument must be a NumPy ndarray or a SciPy sparse matrix.

Note that you will typically use overloaded operators to build and add constraints using matrix semantics. The overloaded `@` operator can be used to build a linear matrix expression or quadratic matrix expression. An overloaded comparison operator can then be used to build a TempConstr object, which is then passed to addConstr.

Arguments:

`Q`: The quadratic constraint matrix - a NumPy 2-D ndarray or a SciPy sparse matrix.

`c`: The linear constraint vector - a NumPy 1-D ndarray. This can be `None` if there are no linear terms.

`sense`: Constraint sense. Valid values are $<$, $>$, or $=$.

`rhs`: Right-hand-side value.

xQ_L: Decision variables for quadratic terms; left multiplier for Q. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the size of the first dimension of `Q`.

xQ_R: Decision variables for quadratic terms; right multiplier for Q. The length of the argument must match the size of the second dimension of `Q`.

xc: Decision variables for linear terms. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the length of `c`.

name: Name for new constraint.

Return value:

The QConstr object.

Example usage:

```
Q = np.full((2, 3), 1)
xL = model.addMVar(2)
xR = model.addMVar(3)
model.addMQConstr(Q, None, '<', 1.0, xL, xR)
```

Model.addMVar()

```
addMVar  ( shape, lb=0.0, ub=GRB.INFINITY, obj=0.0, vtype=GRB.CONTINUOUS,
           name="" )
```

Add an MVar object to a model. An `MVar` is a NumPy ndarray of Gurobi decision variables. The ndarray can have an arbitrary number of dimensions, but you will generally need to slice a multi-dimensional array into 1-D objects to use an `MVar` to build constraints.

You can multiply a 1-D `MVar` by a 2-D matrix (a NumPy dense ndarray or a SciPy sparse matrix), using overloaded Python matrix-multiply operators (`@`), to create a linear matrix expression or quadratic matrix expression, which can then be used to build linear or quadratic objectives or constraints

Note that the returned MVar object supports standard NumPy slicing.

Arguments:

shape: The `shape` of the array.

lb (optional): Lower bound(s) for new variables.

ub (optional): Upper bound(s) for new variables.

obj (optional): Objective coefficient(s) for new variables.

vtype (optional): Variable type(s) for new variables.

name (optional): Names for new variables. The given name will be subscripted by the index of the generator expression, so if the index is an integer, `c` would become `c[0]`, `c[1]`, etc. Note that the generated names will be stored as ASCII strings, so you should avoid using names that contain non-ASCII characters. In addition, names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New `MVar` object.

Example usage:

```
x = model.addMVar(10)                       # add a 1-D array of 10 variables
y = model.addMVar((3,4), vtype=GRB.BINARY) # add a 3x4 2-D array of binary variables
```

```
    print(y[:,1:3])                          # take a slice of a 2-D array
```

Model.addQConstr()

```
addQConstr  ( lhs, sense=None, rhs=None, name="" )
```

Add a quadratic constraint to a model.

Important note: Gurobi can handle both convex and non-convex quadratic constraints. The differences between them can be both important and subtle. Refer to this discussion for additional information.

Arguments:

lhs: Left-hand side for new quadratic constraint. Can be a constant, a Var, a LinExpr, or a QuadExpr.

sense: Sense for new quadratic constraint (GRB.LESS_EQUAL or GRB.GREATER_EQUAL).

rhs: Right-hand side for new quadratic constraint. Can be a constant, a Var, a LinExpr, or a QuadExpr.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New quadratic constraint object.

Example usage:

```
model.addQConstr(x*x + y*y, GRB.LESS_EQUAL, z*z, "c0")
model.addQConstr(x*x + y*y <= 2.0, "c1")
```

Model.addRange()

```
addRange  ( expr, lower, upper, name="" )
```

Add a range constraint to a model. A range constraint states that the value of the input expression must be between the specified `lower` and `upper` bounds in any solution.

Note that range constraints are stored internally as equality constraints. We add an extra variable to the model to capture the range information. Thus, the Sense attribute on a range constraint will always be GRB.EQUAL.

Arguments:

expr: Linear expression for new range constraint. Can be a Var or a LinExpr.

lower: Lower bound for linear expression.

upper: Upper bound for linear expression.

name: Name for new constraint. Note that **name** will be stored as an ASCII string. Thus, a name like 'A→B' will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New constraint object.

Example usage:

```
# 1 <= x + y <= 2
model.addRange(x + y, 1.0, 2.0, "range0")

# overloaded forms
model.addConstr(x + y == [1.0, 2.0], name="range0")
```

Model.addSOS()

```
addSOS ( type, vars, wts=None )
```

Add an SOS constraint to the model. Please refer to this section for details on SOS constraints.

Arguments:

`type`: SOS type (can be `GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`).

`vars`: List of variables that participate in the SOS constraint.

`weights (optional)`: Weights for the variables in the SOS constraint. Default weights are 1, 2, ...

Return value:

New SOS object.

Example usage:

```
model.addSOS(GRB.SOS_TYPE1, [x, y, z], [1, 2, 4])
```

Model.addVar()

```
addVar ( lb=0.0, ub=GRB.INFINITY, obj=0.0, vtype=GRB.CONTINUOUS, name="",
         column=None )
```

Add a decision variable to a model.

Arguments:

`lb (optional)`: Lower bound for new variable.

`ub (optional)`: Upper bound for new variable.

`obj (optional)`: Objective coefficient for new variable.

`vtype (optional)`: Variable type for new variable (`GRB.CONTINUOUS`, `GRB.BINARY`, `GRB.INTEGER`, `GRB.SEMICONT`, or `GRB.SEMIINT`).

`name (optional)`: Name for new variable. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

`column (optional)`: Column object that indicates the set of constraints in which the new variable participates, and the associated coefficients.

Return value:

New variable object.

Example usage:

```
x = model.addVar()                                 # all default arguments
y = model.addVar(vtype=GRB.INTEGER, obj=1.0, name="y") # arguments by name
z = model.addVar(0.0, 1.0, 1.0, GRB.BINARY, "z")      # arguments by position
```

Model.addVars()

```
addVars ( *indices, lb=0.0, ub=GRB.INFINITY, obj=0.0, vtype=GRB.CONTINUOUS,
          name="" )
```

Add multiple decision variables to a model.

Returns a Gurobi tupledict object that contains the newly created variables. The keys for the `tupledict` are derived from the `indices` argument(s). The arguments for this method can take several different forms, which will be described now.

The first arguments provide the indices that will be used as keys to access the variables in the returned `tupledict`. In its simplest version, you would specify one or more integer values, and this method would create the equivalent of a multi-dimensional array of variables. For example, `x = model.addVars(2, 3)` would create six variables, accessed as `x[0,0]`, `x[0,1]`, `x[0,2]`, `x[1,0]`, `x[1,1]`, and `x[1,2]`.

In a more complex version, you can specify arbitrary lists of immutable objects, and this method will create variables for each member of the cross product of these lists. For example, `x = model.addVars([3, 7], ['a', 'b', 'c'])` would create six variables, accessed as `x[3,'a']`, `x[7,'c']`, etc.

You can also provide your own list of tuples as indices. For example, `x = model.addVars([(3,'a'), (3,'b'), (7,'b'), (7,'c')])` would be accessed in the same way as the previous example (`x[3,'a']`, `x[7,'c']`, etc.), except that not all combinations will be present. This is typically how sparse indexing is handled.

Note that while the indices can be provided as multiple lists of objects, or as a list of tuples, the member values for a specific index must always be scalars (`int`, `float`, `string`, ...). For example, `x = model.addVars([(1, 3), 7], ['a'])` is not allowed, since the first argument for the first member would be `(1, 3)`. Similarly, `x = model.addVars([((1, 3),'a'), (7,'a')])` is also not allowed.

The named arguments (`lb`, `obj`, etc.) can take several forms. If you provide a scalar value (or use the default), then every variable will use that value. Thus, for example, `lb=1.0` will give every created variable a lower bound of 1.0. Note that a scalar value for the name argument has a special meaning, which will be discussed separately.

You can also provide a Python `dict` as the argument. In that case, the value for each variable will be pulled from the dict, using the indices argument to build the keys. For example, if the variables created by this method are indexed as `x[i,j]`, then the `dict` provided for the argument should have an entry for each possible `(i,j)` value.

Finally, if your `indices` argument is a single list, you can provide a Python `list` of the same length for the named arguments. For each variable, it will pull the value from the corresponding position in the list.

As noted earlier, the `name` argument is special. If you provide a scalar argument for the name, that argument will be transformed to have a subscript that corresponds to the index of the associated variable. For example, if you do `x = model.addVars(2,3,name="x")`, the variables will get names `x[0,0]`, `x[0,1]`, etc.

Arguments:

`indices`: Indices for accessing the new variables.

`lb (optional)`: Lower bound(s) for new variables.

`ub (optional)`: Upper bound(s) for new variables.

obj (optional): Objective coefficient(s) for new variables.
vtype (optional): Variable type(s) for new variables.
name (optional): Names for new variables. The given name will be subscripted by the index of the generator expression, so if the index is an integer, `c` would become `c[0]`, `c[1]`, etc. Note that the generated names will be stored as ASCII strings, so you should avoid using names that contain non-ASCII characters. In addition, names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Return value:

New tupledict object that contains the new variables as values, using the provided indices as keys.

Example usage:

```
# 3-D array of binary variables
x = model.addVars(3, 4, 5, vtype=GRB.BINARY)

# variables index by tuplelist
l = tuplelist([(1, 2), (1, 3), (2, 3)])
y = model.addVars(l, ub=[1, 2, 3])
```

Model.cbCut()

```
cbCut  ( lhs, sense, rhs )
```

Add a new cutting plane to a MIP model from within a callback function. Note that this method can only be invoked when the `where` value on the callback function is equal to `GRB.Callback.MIPNODE` (see the Callback Codes section for more information).

Cutting planes can be added at any node of the branch-and-cut tree. However, they should be added sparingly, since they increase the size of the relaxation model that is solved at each node and can significantly degrade node processing speed.

Cutting planes are typically used to cut off the current relaxation solution. To retrieve the relaxation solution at the current node, you should first call cbGetNodeRel.

When adding your own cuts, you must set parameter PreCrush to value 1. This setting shuts off a few presolve reductions that sometimes prevent cuts on the original model from being applied to the presolved model.

One very important note: you should only add cuts that are implied by the constraints in your model. If you cut off an integer solution that is feasible according to the original model constraints, *you are likely to obtain an incorrect solution to your MIP problem.*

Arguments:

lhs: Left-hand side for new cut. Can be a constant, a Var, or a LinExpr.
sense: Sense for new cut (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_EQUAL`).
rhs: Right-hand side for new cut. Can be a constant, a Var, or a LinExpr.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    status = model.cbGet(GRB.Callback.MIPNODE_STATUS)
    if status == GRB.OPTIMAL:
      rel = model.cbGetNodeRel([model._vars[0], model._vars[1]])
```

```
        if rel[0] + rel[1] > 1.1:
          model.cbCut(model._vars[0] + model._vars[1] <= 1)

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbGet()

`cbGet ( what )`

Query the optimizer from within the user callback.

Arguments:

what: Integer code that indicates what type of information is being requested by the callback. The set of valid codes depends on the where value that is passed into the user callback function. Please refer to the Callback Codes section for a list of possible where and what values.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.SIMPLEX:
    print(model.cbGet(GRB.Callback.SPX_OBJVAL))

model.optimize(mycallback)
```

Model.cbGetNodeRel()

`cbGetNodeRel ( vars )`

Retrieve values from the node relaxation solution at the current node. Note that this method can only be invoked when the where value on the callback function is equal to GRB.Callback.MIPNODE, and GRB.Callback.MIPNODE_STATUS is equal to GRB.OPTIMAL (see the Callback Codes section for more information).

Arguments:

vars: The variables whose relaxation values are desired. Can be a variable, a matrix variable, a list of variables or matrix variables, or a dict of variables.

Return value:

Values for the specified variables in the node relaxation for the current node. The format will depend on the input argument (a scalar, an ndarray, a list of values or ndarrays, or a dict).

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    print(model.cbGetNodeRel(model._vars))

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbGetSolution()

`cbGetSolution ( vars )`

Retrieve values from the new MIP solution. Note that this method can only be invoked when the `where` value on the callback function is equal to `GRB.Callback.MIPSOL` or `GRB.Callback.MULTIOBJ` (see the Callback Codes section for more information).

Arguments:

`vars`: The variables whose solution values are desired. Can be a variable, a matrix variable, a list of variables or matrix variables, or a dict of variables.

Return value:

Values for the specified variables in the solution. The format will depend on the input argument (a scalar, an ndarray, a list of values or ndarrays, or a dict).

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPSOL:
    print(model.cbGetSolution(model._vars))

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbLazy()

`cbLazy ( lhs, sense, rhs )`

Add a new lazy constraint to a MIP model from within a callback function. Note that this method can only be invoked when the `where` value on the callback function is `GRB.Callback.MIPNODE` or `GRB.Callback.MIPSOL` (see the Callback Codes section for more information).

Lazy constraints are typically used when the full set of constraints for a MIP model is too large to represent explicitly. By only including the constraints that are actually violated by solutions found during the branch-and-cut search, it is sometimes possible to find a proven optimal solution while only adding a fraction of the full set of constraints.

You would typically add a lazy constraint by first querying the current node solution (by calling cbGetSolution from a `GRB.CB_MIPSOL` callback, or cbGetNodeRel from a `GRB.CB_MIPNODE` callback), and then calling `cbLazy()` to add a constraint that cuts off the solution. Gurobi guarantees that you will have the opportunity to cut off any solutions that would otherwise be considered feasible.

Your callback should be prepared to cut off solutions that violate any of your lazy constraints, including those that have already been added. Node solutions will usually respect previously added lazy constraints, but not always.

Note that you must set the LazyConstraints parameter if you want to use lazy constraints.

Arguments:

`lhs`: Left-hand side for new lazy constraint. Can be a constant, a Var, or a LinExpr.

`sense`: Sense for new lazy constraint (`GRB.LESS_EQUAL`, `GRB.EQUAL`, or `GRB.GREATER_-EQUAL`).

`rhs`: Right-hand side for new lazy constraint. Can be a constant, a Var, or a LinExpr.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPSOL:
    sol = model.cbGetSolution([model._vars[0], model._vars[1]])
    if sol[0] + sol[1] > 1.1:
      model.cbLazy(model._vars[0] + model._vars[1] <= 1)

model._vars = model.getVars()
model.optimize(mycallback)
```

Model.cbSetSolution()

`cbSetSolution ( vars, solution )`

Import solution values for a heuristic solution. Only available when the `where` value on the callback function is equal to `GRB.CB_MIPNODE`. (see the Callback Codes section for more information).

When you specify a heuristic solution from a callback, variables initially take undefined values. You should use this method to specify variable values. You can make multiple calls to `cbSetSolution` from one callback invocation to specify values for multiple sets of variables. After the callback, if values have been specified for any variables, the Gurobi optimizer will try to compute a feasible solution from the specified values, possibly filling in values for variables whose values were left undefined. You can also optionally call cbUseSolution within your callback function to try to immediately compute a feasible solution from the specified values.

Arguments:

`vars`: The variables whose values are being set. This can be a list of variables or a single variable.

`solution`: The desired values of the specified variables in the new solution.

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    model.cbSetSolution(vars, newsolution)

model.optimize(mycallback)
```

Model.cbStopOneMultiObj()

`cbStopOneMultiObj ( objcnt )`

Interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process. Only available for multi-objective MIP models and when the `where` member variable is not equal to `GRB.Callback.MULTIOBJ` (see the Callback Codes section for more information).

You would typically stop a multi-objective optimization step by querying the last finished number of multi-objectives steps, and using that number to stop the current step and move on to the next hierarchical objective (if any) as shown in the following example:

Example usage:

```
import time

def mycallback(model, where):
  if where == GRB.Callback.MULTIOBJ:
    # get current objective number
    model._objcnt = model.cbGet(GRB.Callback.MULTIOBJ_OBJCNT)

    # reset start time to current time
    model._starttime = time.time()

  # See if we want to stop current multiobjective step
  else if time.time() - model._starttime > BIG or
          solution is good enough:
    # stop only this optimization step
    cbStopOneMultiObj(model._objcnt)

model._objcnt = 0
model._starttime = time.time()
model.optimize(mycallback)
```

You should refer to the section on Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Arguments:

`objnum`: The number of the multi-objective optimization step to interrupt. For processes running locally, this argument can have the special value -1, meaning to stop the current step.

Model.cbUseSolution()

`cbUseSolution` ( )

Once you have imported solution values using cbSetSolution, you can optionally call `cbUseSolution` to immediately use these values to try to compute a heuristic solution.

Return value:

The objective value for the solution obtained from your solution values (or GRB.INFINITY if no improved solution is found).

Example usage:

```
def mycallback(model, where):
  if where == GRB.Callback.MIPNODE:
    model.cbSetSolution(vars, newsolution)
    objval = model.cbUseSolution()

model.optimize(mycallback)
```

Model.chgCoeff()

```
chgCoeff ( constr, var, newvalue )
```

Change one coefficient in the model. The desired change is captured using a `Var` object, a `Constr` object, and a desired coefficient for the specified variable in the specified constraint. If you make multiple changes to the same coefficient, the last one will be applied.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

Arguments:

`constr`: Constraint for coefficient to be changed.
`var`: Variable for coefficient to be changed.
`newvalue`: Desired new value for coefficient.

Example usage:

```
model.chgCoeff(c0, x, 2.0)
```

Model.computeIIS()

```
computeIIS ( void )
```

Compute an Irreducible Inconsistent Subsystem (IIS). An IIS is a subset of the constraints and variable bounds with the following properties:

- the subsystem represented by the IIS is infeasible, and
- if any of the constraints or bounds of the IIS is removed, the subsystem becomes feasible.

Note that an infeasible model may have multiple IISs. The one returned by Gurobi is not necessarily the one with minimum cardinality; there may exist others with fewer constraints or bounds.

If an IIS computation is interrupted before completion, Gurobi will return the smallest IIS found to that point.

This method populates the `IISCONSTR`, `IISQCONSTR`, and `IISGENCONSTR` constraint attributes, the `IISSOS` SOS attribute, and the `IISLB`, and `IISUB` variable attributes. You can also obtain information about the results of the IIS computation by writing an `.ilp` format file (see Model.write). This file contains only the IIS from the original model.

Note that this method can be used to compute IISs for both continuous and MIP models.

Example usage:

```
model.computeIIS()
model.write("model.ilp")
```

Model.copy()

```
copy ( )
```

Copy a model. Note that due to the lazy update approach in Gurobi, you have to call update before copying it.

Return value:

Copy of model.

Example usage:

```
model.update() # If you have unstaged changes in the model
copy = model.copy()
```

Model.discardConcurrentEnvs()

```
discardConcurrentEnvs ( )
```

Discard concurrent environments for a model.

The concurrent environments created by getConcurrentEnv will be used by every subsequent call to the concurrent optimizer until the concurrent environments are discarded.

Example usage:

```
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

env0.setParam('Method', 0)
env1.setParam('Method', 1)

model.optimize()

model.discardConcurrentEnvs()
```

Model.discardMultiobjEnvs()

```
discardMultiobjEnvs ( )
```

Discard all multi-objective environments associated with the model, thus restoring multi objective optimization to its default behavior.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Use getMultiobjEnv to create a multi-objective environment.

Example usage:

```
env0 = model.getMultiobjEnv(0)
env1 = model.getMultiobjEnv(1)

env0.setParam('Method', 2)
env1.setParam('Method', 1)

model.optimize()

model.discardMultiobjEnvs()
```

Model.dispose()

```
dispose ( )
```

Free all resources associated with this Model object. After this method is called, this Model object must no longer be used.

Example usage:

```
env = Env()
model = read("misc07.mps", env)
model.optimize()
model.dispose()
env.dispose()
```

Model.feasRelaxS()

```
feasRelaxS  ( relaxobjtype, minrelax, vrelax, crelax )
```

Modifies the `Model` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution. Note also that this is a simplified version of this method - use feasRelax for more control over the relaxation performed.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the magnitudes of the bound and constraint violations.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the sum of the squares of the bound and constraint violations.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the total number of bound and constraint violations.

To give an example, if a constraint is violated by 2.0, it would contribute `2.0` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2.0*2.0` for `relaxobjtype=1`, and it would contribute `1.0` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=False`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=True`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelaxS` must solve an optimization problem to find the minimum possible relaxation when `minrelax=True`, which can be quite expensive.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use copy to create a copy before invoking this method.

Arguments:

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.
`minrelax`: The type of feasibility relaxation to perform.
`vrelax`: Indicates whether variable bounds can be relaxed.
`crelax`: Indicates whether constraints can be relaxed.

Return value:

Zero if `minrelax` is False. If `minrelax` is True, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

Example usage:

```
if model.status == GRB.INFEASIBLE:
  model.feasRelaxS(1, False, False, True)
  model.optimize()
```

Model.feasRelax()

```
feasRelax  ( relaxobjtype, minrelax, vars, lbpen, ubpen, constrs, rhspen )
```

Modifies the `Model` object to create a feasibility relaxation. Note that you need to call optimize on the result to compute the actual relaxed solution. Note also that this is a more complex version of this method - use feasRelaxS for a simplified version.

The feasibility relaxation is a model that, when solved, minimizes the amount by which the solution violates the bounds and linear constraints of the original model. This method provides a number of options for specifying the relaxation.

If you specify `relaxobjtype=0`, the objective of the feasibility relaxation is to minimize the sum of the weighted magnitudes of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost per unit violation in the lower bounds, upper bounds, and linear constraints, respectively.

If you specify `relaxobjtype=1`, the objective of the feasibility relaxation is to minimize the weighted sum of the squares of the bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the coefficients on the squares of the lower bound, upper bound, and linear constraint violations, respectively.

If you specify `relaxobjtype=2`, the objective of the feasibility relaxation is to minimize the weighted count of bound and constraint violations. The `lbpen`, `ubpen`, and `rhspen` arguments specify the cost of violating a lower bound, upper bound, and linear constraint, respectively.

To give an example, if a constraint with `rhspen` value `p` is violated by 2.0, it would contribute `2*p` to the feasibility relaxation objective for `relaxobjtype=0`, it would contribute `2*2*p` for `relaxobjtype=1`, and it would contribute `p` for `relaxobjtype=2`.

The `minrelax` argument is a boolean that controls the type of feasibility relaxation that is created. If `minrelax=False`, optimizing the returned model gives a solution that minimizes the cost of the violation. If `minrelax=True`, optimizing the returned model finds a solution that minimizes the original objective, but only from among those solutions that minimize the cost of the violation. Note that `feasRelax` must solve an optimization problem to find the minimum possible relaxation when `minrelax=True`, which can be quite expensive.

Note that this is a destructive method: it modifies the model on which it is invoked. If you don't want to modify your original model, use copy to create a copy before invoking this method.

Arguments:

`relaxobjtype`: The cost function used when finding the minimum cost relaxation.

`minrelax`: The type of feasibility relaxation to perform.

`vars`: Variables whose bounds are allowed to be violated.

`lbpen`: Penalty for violating a variable lower bound. One entry for each variable in argument `vars`.

`ubpen`: Penalty for violating a variable upper bound. One entry for each variable in argument `vars`.

`constrs`: Linear constraints that are allowed to be violated.

rhspen: Penalty for violating a linear constraint. One entry for each constraint in argument `constrs`.

Return value:

Zero if `minrelax` is False. If `minrelax` is True, the return value is the objective value for the relaxation performed. If the value is less than 0, it indicates that the method failed to create the feasibility relaxation.

Example usage:

```
if model.status == GRB.INFEASIBLE:
  vars = model.getVars()
  ubpen = [1.0]*model.numVars
  model.feasRelax(1, False, vars, None, ubpen, None, None)
  model.optimize()
```

Model.fixed()

```
fixed ( )
```

Create the fixed model associated with a MIP model. The MIP model must have a solution loaded (e.g., after a call to the optimize method). In the fixed model, each integer variable is fixed to the value that variable takes in the MIP solution. In addition, continuous variables may be fixed to satisfy SOS or general constraints. The result is a model without any integrality constraints, SOS constraints, or general constraints.

Note that, while the fixed problem is always a continuous model, it may contain a non-convex quadratic objective or non-convex quadratic constraints. As a result, it may still be solved using the MIP algorithm.

Return value:

Fixed model associated with calling object.

Example usage:

```
fixed = model.fixed()
```

Model.getA()

```
getA ( )
```

Query the linear constraint matrix of the model. You'll need to have `scipy` installed for this function to work.

Arguments:

Return value:

The matrix as a `scipy.sparse` matrix in CSR format.

Example usage:

```
A = model.getA()
print(A.toarray())
```

Model.getAttr()

```
getAttr ( attrname, objs=None )
```

Query the value of an attribute. When called with a single argument, it returns the value of a model attribute. When called with two arguments, it returns the value of an attribute for either a list or a dictionary containing either variables or constraints. If called with a list, the result is a list. If called with a dictionary, the result is a dictionary that uses the same keys, but is populated with the requested attribute values. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

Arguments:

`attrname`: Name of the attribute.

`objs (optional)`: List or dictionary containing either constraints or variables

Example usage:

```
print(model.numintvars)
print(model.getAttr("numIntVars"))
print(model.getAttr(GRB.Attr.NumIntVars))
print(model.getAttr("X", model.getVars()))
print(model.getAttr("Pi", model.getConstrs()))
```

Model.getCoeff()

```
getCoeff ( constr, var )
```

Query the coefficient of variable `var` in linear constraint `constr` (note that the result can be zero).

Arguments:

`constr`: The requested constraint.

`var`: The requested variable.

Return value:

The current value of the requested coefficient.

Example usage:

```
print(model.getCoeff(constr, var))
```

Model.getCol()

```
getCol ( var )
```

Retrieve the list of constraints in which a variable participates, and the associated coefficients. The result is returned as a Column object.

Arguments:

`var`: The variable of interest.

Return value:

A Column object that captures the set of constraints in which the variable participates.

Example usage:

```
print(model.getCol(x))
```

Model.getConcurrentEnv()

```
getConcurrentEnv ( num )
```

Create/retrieve a concurrent environment for a model.

This method provides fine-grained control over the concurrent optimizer. By creating your own concurrent environments and setting appropriate parameters on these environments (e.g., the Method parameter), you can control exactly which strategies the concurrent optimizer employs. For example, if you create two concurrent environments, and set `Method` to primal simplex for one and dual simplex for the other, subsequent concurrent optimizer runs will use the two simplex algorithms rather than the default choices.

Note that you must create contiguously numbered concurrent environments, starting with `num=0`. For example, if you want three concurrent environments, they must be numbered 0, 1, and 2.

Once you create concurrent environments, they will be used for every subsequent concurrent optimization on that model. Use discardConcurrentEnvs to revert back to default concurrent optimizer behavior.

Arguments:

`num (int)`: The concurrent environment number.

Return value:

The concurrent environment for the model.

Example usage:

```
env0 = model.getConcurrentEnv(0)
env1 = model.getConcurrentEnv(1)

env0.setParam('Method', 0)
env1.setParam('Method', 1)

model.optimize()

model.discardConcurrentEnvs()
```

Model.getConstrByName()

```
getConstrByName ( name )
```

Retrieve a linear constraint from its name. If multiple linear constraints have the same name, this method chooses one arbitrarily.

Arguments:

`name`: Name of desired constraint.

Return value:

Constraint with the specified name.

Example usage:

```
c0 = model.getConstrByName("c0")
```

Model.getConstrs()

```
getConstrs ( )
```

Retrieve a list of all linear constraints in the model.

Return value:

All linear constraints in the model.

Example usage:

```
constrs = model.getConstrs()
c0 = constrs[0]
```

Model.getGenConstrMax()

```
getGenConstrMax ( genconstr )
```

Retrieve the data associated with a general constraint of type MAX. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrMax for a description of the semantics of this general constraint type.

Arguments:

`genconstr`: The general constraint object of interest.

Return value:

A tuple (resvar, vars, constant) that contains the data of the general constraint:

`resvar (Var)`: Resultant variable of the MAX constraint.

`vars (list of Var)`: Operand variables of the MAX constraint.

`constant (float)`: Additional constant operand of the MAX constraint.

Example usage:

```
(resvar, vars, constant) = model.getGenConstrMax(model.getGenConstrs()[0])
```

Model.getGenConstrMin()

```
getGenConstrMin ( genconstr )
```

Retrieve the data associated with a general constraint of type MIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrMin for a description of the semantics of this general constraint type.

Arguments:

`genconstr`: The general constraint object of interest.

Return value:

A tuple (resvar, vars, constant) that contains the data of the general constraint:

`resvar (Var)`: Resultant variable of the MIN constraint.

`vars (list of Var)`: Operand variables of the MIN constraint.

`constant (float)`: Additional constant operand of the MIN constraint.

Example usage:

```
(resvar, vars, constant) = model.getGenConstrMin(model.getGenConstrs()[0])
```

Model.getGenConstrAbs()

```
getGenConstrAbs ( genconstr )
```

Retrieve the data associated with a general constraint of type ABS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrAbs for a description of the semantics of this general constraint type.

Arguments:

`genconstr`: The general constraint object of interest.

Return value:

A tuple (resvar, argvar) that contains the data of the general constraint:

`resvar (Var)`: Resultant variable of ABS constraint.

`argvar (Var)`: Argument variable of ABS constraint.

Example usage:

```
(resvar, argvar) = model.getGenConstrAbs(model.getGenConstrs()[0])
```

Model.getGenConstrAnd()

```
getGenConstrAnd ( genconstr )
```

Retrieve the data associated with a general constraint of type AND. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrAnd for a description of the semantics of this general constraint type.

Arguments:

`genconstr`: The general constraint object of interest.

Return value:

A tuple (resvar, vars) that contains the data of the general constraint:

`resvar (Var)`: Resultant variable of AND constraint.

`vars (list of Var)`: Operand variables of AND constraint.

Example usage:

```
(resvar, vars) = model.getGenConstrAnd(model.getGenConstrs()[0])
```

Model.getGenConstrOr()

```
getGenConstrOr ( genconstr )
```

Retrieve the data associated with a general constraint of type OR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrOr for a description of the semantics of this general constraint type.

Arguments:

`genconstr`: The general constraint object of interest.

Return value:

A tuple (resvar, vars) that contains the data of the general constraint:

`resvar (Var)`: Resultant variable of OR constraint.

vars (list of Var): Operand variables of OR constraint.

Example usage:

```
(resvar, vars) = model.getGenConstrOr(model.getGenConstrs()[0])
```

Model.getGenConstrIndicator()

getGenConstrIndicator (genconstr)

Retrieve the data associated with a general constraint of type INDICATOR. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrIndicator for a description of the semantics of this general constraint type.

Arguments:

genconstr: The general constraint object of interest.

Return value:

A tuple (binvar, binval, expr, sense, rhs) that contains the data of the general constraint:

binvar (Var): Antecedent variable of indicator constraint.

binval (Boolean): Value of antecedent variable that activates the linear constraint.

expr (LinExpr): LinExpr object containing the left-hand side of the constraint triggered by the indicator.

sense (char): Sense of linear constraint triggered by the indicator (e.g., `GRB.LESS_EQUAL`).

rhs (float): Right-hand side of linear constraint triggered by the indicator.

Example usage:

```
(binvar, binval, expr, sense, rhs) =
    model.getGenConstr(model.getGenConstrIndicator()[3])
```

Model.getGenConstrPWL()

getGenConstrPWL (genconstr)

Retrieve the data associated with a general constraint of type PWL. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPWL for a description of the semantics of this general constraint type.

Arguments:

genc: The general constraint object.

Return value:

A tuple (xvar, yvar, xpts, ypts) that contains the data of the general constraint:

xvar (Var): The x variable.

yvar (Var): The y variable.

xpts (list of float): The x values for the points that define the piecewise-linear function.

ypts (list of float): The y values for the points that define the piecewise-linear function.

Example usage:

```
(xvar, yvar, xpts, ypts) = model.getGenConstrPWL(model.getGenConstrs()[0])
```

Model.getGenConstrPoly()

`getGenConstrPoly ( genconstr )`

Retrieve the data associated with a general constraint of type POLY. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPoly for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar, p) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`p (list of float)`: The coefficients for polynomial function.

Example usage:

```
(xvar, yvar, p) = model.getGenConstrPoly(model.getGenConstrs()[0])
```

Model.getGenConstrExp()

`getGenConstrExp ( genconstr )`

Retrieve the data associated with a general constraint of type EXP. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExp for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

Example usage:

```
(xvar, yvar) = model.getGenConstrExp(model.getGenConstrs()[0])
```

Model.getGenConstrExpA()

`getGenConstrExpA ( genconstr )`

Retrieve the data associated with a general constraint of type EXPA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrExpA for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar, a) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`a (float)`: The base of the function.

Example usage:

```
(xvar, yvar, a) = model.getGenConstrExpA(model.getGenConstrs()[0])
```

Model.getGenConstrLog()

```
getGenConstrLog ( genconstr )
```

Retrieve the data associated with a general constraint of type LOG. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLog for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

Model.getGenConstrLogA()

```
getGenConstrLogA ( genconstr )
```

Retrieve the data associated with a general constraint of type LOGA. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrLogA for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar, a) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`a (float)`: The base of the function.

Model.getGenConstrPow()

```
getGenConstrPow ( genconstr )
```

Retrieve the data associated with a general constraint of type POW. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrPow for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar, a) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

`a (float)`: The exponent of the function.

Model.getGenConstrSin()

```
getGenConstrSin  ( genconstr )
```

Retrieve the data associated with a general constraint of type SIN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrSin for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

Model.getGenConstrCos()

```
getGenConstrCos  ( genconstr )
```

Retrieve the data associated with a general constraint of type COS. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrCos for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

Model.getGenConstrTan()

```
getGenConstrTan  ( genconstr )
```

Retrieve the data associated with a general constraint of type TAN. Calling this method for a general constraint of a different type leads to an exception. You can query the GenConstrType attribute to determine the type of the general constraint.

See also addGenConstrTan for a description of the semantics of this general constraint type.

Arguments:

`genc`: The general constraint object.

Return value:

A tuple (xvar, yvar) that contains the data of the general constraint:

`xvar (Var)`: The x variable.

`yvar (Var)`: The y variable.

Model.getGenConstrs()

`getGenConstrs ( )`

Retrieve a list of all general constraints in the model.

Return value:

All general constraints in the model.

Example usage:

```
gencons = model.getGenConstrs()
for gc in gencons:
  print(model.getGenConstr(gc))
```

Model.getJSONSolution()

`getJSONSolution ( )`

After a call to optimize, this method returns the resulting solution and related model attributes as a JSON string. Please refer to the JSON solution format section for details.

Return value:

A JSON string.

Example usage:

```
model = read('p0033.mps')
model.optimize()
print(model.getJSONSolution())
```

Model.getMultiobjEnv()

`getMultiobjEnv ( index )`

Create/retrieve a multi-objective environment for the objective with the given index. This environment enables fine-grained control over the multi-objective optimization process. Specifically, by changing parameters on this environment, you modify the behavior of the optimization that occurs during the corresponding pass of the multi-objective optimization.

Each multi-objective environment starts with a copy of the current model environment.

Please refer to the discussion of Multiple Objectives for information on how to specify multiple objective functions and control the trade-off between them.

Use discardMultiobjEnvs to discard multi-objective environments and return to standard behavior.

Arguments:

`index (int)`: The objective index.

Return value:

The multi-objective environment for the model.

Example usage:

```
env0 = model.getMultiobjEnv(0)
env1 = model.getMultiobjEnv(1)

env0.setParam('TimeLimit', 100)
env1.setParam('TimeLimit', 10)

model.optimize()

model.discardMultiobjEnvs()
```

Model.getObjective()

getObjective (index=None)

Retrieve the model objective(s).

Call this with no argument to retrieve the primary objective, or with an integer argument to retrieve the corresponding alternative objective.

Arguments:

index (int, optional): The index for the requested alternative objective.

Return value:

The model objective. A LinExpr object for a linear objective, or a QuadExpr object for a quadratic objective. Note that alternative objectives are always linear.

Example usage:

```
obj = model.getObjective()
print(obj.getValue())
```

Model.getParamInfo()

getParamInfo (paramname)

Retrieve information about a Gurobi parameter, including the type, the current value, the minimum and maximum allowed values, and the default value.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Arguments:

paramname: String containing the name of the parameter of interest. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and the method returns None.

Return value:

Returns a 6-entry tuple that contains: the parameter name, the parameter type, the current value, the minimum value, the maximum value, and the default value.

Example usage:

```
print(model.getParamInfo('Heuristics'))
```

Model.getPWLObj()

```
getPWLObj ( var )
```

Retrieve the piecewise-linear objective function for a variable. The function returns a list of tuples, where each provides the x and y coordinates for the points that define the piecewise-linear objective function.

Refer to this discussion for additional information on what the values in x and y mean.

Arguments:

`var`: A Var object that gives the variable whose objective function is being retrieved.

Return value:

The points that define the piecewise-linear objective function.

Example usage:

```
> print(model.getPWLObj(var))
[(1, 1), (2, 2), (3, 4)]
```

Model.getQConstrs()

```
getQConstrs ( )
```

Retrieve a list of all quadratic constraints in the model.

Return value:

All quadratic constraints in the model.

Example usage:

```
qconstrs = model.getQConstrs()
qc0 = qconstrs[0]
```

Model.getQCRow()

```
getQCRow ( qconstr )
```

Retrieve the left-hand side expression from a quadratic constraint. The result is returned as a QuadExpr object.

Arguments:

`qconstr`: The constraint of interest.

Return value:

A QuadExpr object that captures the left-hand side of the quadratic constraint.

Example usage:

```
print(model.getQCRow(model.getQConstrs()[0]))
```

Model.getRow()

```
getRow ( constr )
```

Retrieve the list of variables that participate in a constraint, and the associated coefficients. The result is returned as a LinExpr object.

Arguments:

`constr`: The constraint of interest.

Return value:

A LinExpr object that captures the set of variables that participate in the constraint.

Example usage:

```
print(model.getRow(c0))
```

Model.getSOS()

```
getSOS ( sos )
```

Retrieve information about an SOS constraint. The result is a tuple that contains the SOS type (1 or 2), the list of participating Var objects, and the list of associated SOS weights.

Arguments:

`sos`: The SOS object of interest.

Return value:

A tuple that contains the SOS type (1 or 2), a list of participating Var objects, and a list of associated SOS weights.

Example usage:

```
(sostype, vars, weights) = model.getSOS(s)
```

Model.getSOSs()

```
getSOSs ( )
```

Retrieve a list of all SOS constraints in the model.

Return value:

All SOS constraints in the model.

Example usage:

```
sos = model.getSOSs()
for s in sos:
  print(model.getSOS(s))
```

Model.getTuneResult()

```
getTuneResult ( )
```

Use this routine to retrieve the results of a previous tune call. Calling this method with argument `n` causes tuned parameter set `n` to be copied into the model. Parameter sets are stored in order of decreasing quality, with parameter set 0 being the best. The number of available sets is stored in attribute TuneResultCount.

Once you have retrieved a tuning result, you can call optimize to use these parameter settings to optimize the model, or write to write the changed parameters to a `.prm` file.

Please refer to the parameter tuning section for details on the tuning tool.

Arguments:

`n`: The index of the tuning result to retrieve. The best result is available as index 0. The number of stored results is available in attribute TuneResultCount.

Example usage:

```
model.tune()
for i in range(model.tuneResultCount):
  model.getTuneResult(i)
  model.write('tune'+str(i)+'.prm')
```

Model.getVarByName()

`getVarByName ( name )`

Retrieve a variable from its name. If multiple variables have the same name, this method chooses one arbitrarily.

Arguments:

`name`: Name of desired variable.

Return value:

Variable with the specified name.

Example usage:

```
x0 = model.getVarByName("x0")
```

Model.getVars()

`getVars ( )`

Retrieve a list of all variables in the model.

Return value:

All variables in the model.

Example usage:

```
vars = model.getVars()
x0 = vars[0]
```

Model.message()

`message ( msg )`

Append a string to the Gurobi log file.

Arguments:

`msg`: String to append to Gurobi log file.

Example usage:

```
model.message('New message')
```

Model.optimize()

`optimize ( callback=None )`

Optimize the model. The algorithm used for the optimization depends on the model type (simplex or barrier for a continuous model; branch-and-cut for a MIP model). Upon successful completion, this method will populate the solution related attributes of the model. See the Attributes section for more information on attributes.

Please consult this section for a discussion of some of the practical issues associated with solving a precisely defined mathematical model using finite-precision floating-point arithmetic.

Note that this method will process all pending model modifications.

Arguments:

`callback`: Callback function. The callback function should take two arguments, `model` and `where`. During the optimization, the function will be called periodically, with `model` set to the model being optimized, and `where` indicating where in the optimization the callback is called from. See the Callback class for additional information.

Example usage:

```
model.optimize()
```

Model.optimizeBatch()

```
optimizeBatch ( )
```

Submit a new batch request to the Cluster Manager. Returns the `BatchID` (a string), which uniquely identifies the job in the Cluster Manager and can be used to query the status of this request (from this program or from any other). Once the request has completed, the `BatchID` can also be used to retrieve the associated solution. To submit a batch request, you must tag at least one element of the model by setting one of the VTag, CTag or QCTag attributes. For more details on batch optimization, please refer to the Batch Optimization section.

Note that this routine will process all pending model modifications.

Example usage:

```
# Submit batch request
batchID = model.optimizeBatch()
```

Return value:

A unique string identifier for the batch request.

Model.presolve()

```
presolve ( )
```

Perform presolve on a model.

Return value:

Presolved version of original model.

Example usage:

```
p = model.presolve()
p.printStats()
```

Model.printAttr()

```
printAttr ( attrs, filter='*' )
```

Print the value of one or more attributes. If `attrs` is a constraint or variable attribute, print all non-zero values of the attribute, along with the associate constraint or variable names. If `attrs` is a list of attributes, print attribute values for all listed attributes. The method takes an optional `filter` argument, which allows you to select which specific attribute values to print (by filtering on the constraint or variable name).

See the Attributes section for a list of all available attributes.

Arguments:

`attrs`: Name of attribute or attributes to print. The value can be a single attribute or a list of attributes. If a list is given, all listed attributes must be of the same type (model, variable, or constraint).

`filter (optional)`: Filter for values to print -- name of constr/var must match filter to be printed.

Example usage:

```
model.printAttr('x')          # all non-zero solution values
model.printAttr('lb', 'x*')   # bounds for vars whose names begin with 'x'
model.printAttr(['lb', 'ub']) # lower and upper bounds
```

Model.printQuality()

`printQuality ( )`

Print statistics about the quality of the computed solution (constraint violations, integrality violations, etc.).

For continuous models, the output will include the maximum unscaled and scaled violation, as well as the variable or constraint name associated with the worst unscaled violation. For MIP models, the output will include the maximum unscaled violation and the associated variable or constraint name.

Example usage:

```
model.optimize()
model.printQuality()
```

Model.printStats()

`printStats ( )`

Print statistics about the model (number of constraints and variables, number of non-zeros in constraint matrix, smallest and largest coefficients, etc.).

Example usage:

```
model.printStats()
```

Model.read()

`read ( filename )`

This method is the general entry point for importing data from a file into a model. It can be used to read basis files for continuous models, start vectors for MIP models, or parameter settings.

The type of data read is determined by the file suffix. File formats are described in the File Format section.

Note that this is **not** the method to use if you want to read a new model from a file. For that, use the read command.

Arguments:

`filename`: Name of the file to read. The suffix on the file must be either `.bas` (for an LP basis), `.mst` or `.sol` (for a MIP start), `.hnt` (for MIP hints), `.ord` (for a priority order), or `.prm` (for a parameter file). The suffix may optionally be followed by `.zip`, `.gz`, `.bz2`, or `.7z`. The file name may contain `*` or `?` wildcards. No file is read when no wildcard match is found. If more than one match is found, this method will attempt to read the first matching file.

Example usage:

```
model.read('input.bas')
model.read('input.mst')
```

Model.relax()

```
relax ( )
```

Create the relaxation of a MIP model. Transforms integer variables into continuous variables, and removes SOS and general constraints.

Return value:

Relaxed version of model.

Example usage:

```
r = model.relax()
```

Model.remove()

```
remove ( items )
```

Remove variables, linear constraints, quadratic constraints, SOS constraints, or general constraints from a model.

Arguments:

`items`: The items to remove from the model. Argument can be a single Var, Constr, QConstr, SOS, or GenConstr, or a `list`, `tuple`, or `dict` containing these objects. If the argument is a `dict`, the values will be removed, not the keys.

Example usage:

```
model.remove(model.getVars()[0])
model.remove(model.getVars()[0:10])
model.remove(model.getConstrs()[0])
model.remove(model.getConstrs()[1:3])
model.remove(model.getQConstrs()[0])
model.remove(model.getSOSs()[0])
model.remove(model.getGenConstrs()[0])
```

Model.reset()

```
reset ( clearall=0 )
```

Reset the model to an unsolved state, discarding any previously computed solution information.

Arguments:

`clearall (int, optional)`: Should additional information such as MIP starts, variable hints, branching priorities, lazy flags, and partition information be cleared?

Example usage:

```
model.reset(0)
```

Model.resetParams()

```
resetParams ( )
```

Reset all parameters to their default values.

Example usage:

```
model.resetParams()
```

Model.setAttr()

```
setAttr ( attrname, objects, newvalues )
```

Change the value of an attribute.

Call this method with two arguments (i.e., `setAttr(attrname, newvalue)`) to set a model attribute.

Call it with three arguments (i.e., `setAttr(attrname, objects, newvalues)`) to set attribute values for a list or dict of model objects (`Var` objects, `Constr` objects, etc.). To set the same value for all objects in the second argument, you can pass a scalar value in the third argument. If the second argument is a list, the third argument should be a list of the same length. If the second argument is a dict, the third argument should be dict with a value for every key from the second.

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

Arguments:

`attrname`: Name of attribute to set.

`objs`: List of model objects (Var or Constr or ...)

`newvalue`: Desired new value(s) for attribute.

Example usage:

```
model.setAttr("objCon", 0)
model.setAttr(GRB.Attr.ObjCon, 0)
model.setAttr("LB", model.getVars(), [0]*model.numVars)
model.setAttr("RHS", model.getConstrs(), [1.0]*model.numConstrs)
model.setAttr("vType", model.getVars(), GRB.CONTINUOUS)
model.objcon = 0
```

Model.setMObjective()

```
setMObjective  ( Q, c, constant, xQ_L=None, xQ_R=None, xc=None, sense=None )
```

Set the model objective equal to a quadratic (or linear) expression using matrix semantics.

Note that you will typically use overloaded operators to set the objective using matrix objects. The overloaded `@` operator can be used to build a linear matrix expression or a quadratic matrix expression, which is then passed to setObjective.

Arguments:

`Q`: The quadratic objective matrix - a NumPy 2-D dense ndarray or a SciPy sparse matrix. This can be `None` if there are no quadratic terms.

`c`: The linear constraint vector - a NumPy 1-D ndarray. This can be `None` if there are no linear terms.

`constant`: Objective constant.

`xQ_L (optional)`: Decision variables for quadratic objective terms; left multiplier for Q. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the size of the first dimension of `Q`.

`xQ_R (optional)`: Decision variables for quadratic objective terms; right multiplier for Q. The length of the argument must match the size of the second dimension of `Q`.

`xc (optional)`: Decision variables for linear objective terms. Argument can be an MVar object, a list of Var objects, or `None` (`None` uses all variables in the model). The length of the argument must match the length of `c`.

`sense (optional)`: Optimization sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value to determine the sense.

Example usage:

```
c = np.full(10, 1.0)
xc = model.addMVar(10)

model.setMObjective(None, c, 0.0, None, None, xc, GRB.MAXIMIZE)

Q = np.full((2, 3), 1.0)
xL = model.addMVar(2)
xR = model.addMVar(3)

model.setMObjective(Q, None, 0.0, xL, xR, None, GRB.MINIMIZE)
```

Model.setObjective()

```
setObjective  ( expr, sense=None )
```

Set the model objective equal to a linear or quadratic expression (for multi-objective optimization, see setObjectiveN).

Note that you can also modify a linear model objective using the Obj variable attribute. If you wish to mix and match these two approaches, please note that this method will replace the existing objective.

Arguments:

expr: New objective expression. Argument can be a linear or quadratic expression (an objective of type LinExpr or QuadExpr).

sense (optional): Optimization sense (`GRB.MINIMIZE` for minimization, `GRB.MAXIMIZE` for maximization). Omit this argument to use the `ModelSense` attribute value to determine the sense.

Example usage:

```
model.setObjective(x + y, GRB.MAXIMIZE)
model.setObjective(x*x + y*y)
```

Model.setObjectiveN()

```
setObjectiveN ( expr, index, priority=0, weight=1, abstol=0, reltol=0,
                name="" )
```

Set an alternative optimization objective equal to a linear expression.

Please refer to the discussion of Multiple Objectives for more information on the use of alternative objectives.

Note that you can also modify an alternative objective using the ObjN variable attribute. If you wish to mix and match these two approaches, please note that this method replaces the entire existing objective, while the `ObjN` attribute can be used to modify individual terms.

Arguments:

expr (LinExpr): New alternative objective.

index (int): Index for new objective. If you use an index of 0, this routine will change the primary optimization objective.

priority (int, optional): Priority for the alternative objective. This initializes the ObjNPriority attribute for this objective.

weight (float, optional): Weight for the alternative objective. This initializes the ObjNWeight attribute for this objective.

abstol (float, optional): Absolute tolerance for the alternative objective. This initializes the ObjNAbsTol attribute for this objective.

reltol (float, optional): Relative tolerance for the alternative objective. This initializes the ObjNRelTol attribute for this objective.

name (string, optional): Name of the alternative objective. This initializes the ObjNName attribute for this objective. Note that `name` will be stored as an ASCII string. Thus, a name like `'A→B'` will produce an error, because '→' can not be represented as an ASCII character. Note also that names that contain spaces are strongly discouraged, because they can't be written to LP format files.

Example usage:

```
# Primary objective: x + 2 y
model.setObjectiveN(x + 2*y, 0, 2)
# Alternative, lower priority objectives: 3 y + z and x + z
model.setObjectiveN(3*y + z, 1, 1)
model.setObjectiveN(x + z, 2, 0)
```

Model.setPWLObj()

`setPWLObj ( var, x, y )`

Set a piecewise-linear objective function for a variable.

The arguments to this method specify a list of points that define a piecewise-linear objective function for a single variable. Specifically, the x and y arguments give coordinates for the vertices of the function.

For additional details on piecewise-linear objective functions, refer to this discussion.

Arguments:

`var`: A Var object that gives the variable whose objective function is being set.

`x`: The x values for the points that define the piecewise-linear function. Must be in non-decreasing order.

`y`: The y values for the points that define the piecewise-linear function.

Example usage:

```
model.setPWLObj(var, [1, 3, 5], [1, 2, 4])
```

Model.setParam()

`setParam ( paramname, newvalue )`

Set the value of a parameter to a new value. Note that this method only affects the parameter setting for this model. Use global function setParam to change the parameter for all models.

You can also set parameters using the `Model.Params` class. For example, to set parameter `MIPGap` to value 0 for model `m`, you can do either `m.setParam('MIPGap', 0)` or `m.Params.MIPGap=0`.

Please consult the parameter section for a complete list of Gurobi parameters, including descriptions of their purposes and their minimum, maximum, and default values.

Arguments:

`paramname`: String containing the name of parameter that you would like to modify. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and none are modified. Note that case is ignored.

`newvalue`: Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

Example usage:

```
model.setParam("heu*", 0.5)
model.setParam(GRB.Param.heuristics, 0.5)
model.setParam("heu*", "default")
```

Model.singleScenarioModel()

`singleScenarioModel ( )`

Capture a single scenario from a multi-scenario model. Use the ScenarioNumber parameter to indicate which scenario to capture.

The model on which this method is invoked must be a multi-scenario model, and the result will be a single-scenario model.

Return value:

Model for a single scenario.

Example usage:

```
model.params.ScenarioNumber = 0
s = model.singleScenarioModel()
```

Model.terminate()

```
terminate ( )
```

Generate a request to terminate the current optimization. This method is typically called from within a user callback (see Callbacks for more information). When the optimization stops, the Status attribute will be equal to `GRB_INTERRUPTED`.

Example usage:

```
model.terminate()
```

Model.tune()

```
tune ( )
```

Perform an automated search for parameter settings that improve performance. Upon completion, this method stores the best parameter sets it found. The number of stored parameter sets can be determined by querying the value of the TuneResultCount attribute. The actual settings can be retrieved using getTuneResult

Please refer to the parameter tuning section for details on the tuning tool.

Example usage:

```
model.tune()
```

Model.update()

```
update ( )
```

Process any pending model modifications.

Example usage:

```
model.update()
```

Model.write()

```
write ( filename )
```

This method is the general entry point for writing optimization data to a file. It can be used to write optimization models, solutions vectors, basis vectors, start vectors, or parameter settings. The type of data written is determined by the file suffix. File formats are described in the File Format section.

Note that writing a model to a file will process all pending model modifications. However, writing other model information (solutions, bases, etc.) will not.

Note also that when you write a Gurobi parameter file (PRM), both integer or double parameters not at their default value will be saved, but no string parameter will be saved into the file.

Arguments:

`filename`: The name of the file to be written. The file type is encoded in the file name suffix. Valid suffixes are `.mps`, `.rew`, `.lp`, or `.rlp` for writing the model itself, `.ilp` for writing just the IIS associated with an infeasible model (see Model.computeIIS for further information), `.sol` for writing the current solution, `.mst` for writing a start vector, `.hnt` for writing a hint file, `.bas` for writing an LP basis, `.prm` for writing modified parameter settings, `.attr` for writing model attributes, or `.json` for writing solution information in JSON format. If your system has compression utilities installed (e.g., `7z` or `zip` for Windows, and `gzip`, `bzip2`, or `unzip` for Linux or Mac OS), then the files can be compressed, so additional suffixes of `.gz`, `.bz2`, or `.7z` are accepted.

Example usage:

```
model.write("out.mst")
model.write("out.sol")
```

6.3 Var

Gurobi variable object. Variables are always associated with a particular model. You create a variable object by adding a variable to a model (using Model.addVar), rather than by using a `Var` constructor.

Variable objects have a number of attributes. The full list can be found in the Attributes section of this document. Some variable attributes can only be queried, while others can also be set. Recall that the Gurobi optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about variable attributes. Consider the `lb` attribute. Its value can be queried using `var.lb`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `var.LB`. It can be set using a standard assignment statement (e.g., `var.lb = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `x` attribute), so attempts to assign new values to them will raise an exception.

You can also use Var.getAttr/ Var.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.LB`).

To build expressions using variable objects, you generally use operator overloading. You can build either linear or quadratic expressions:

```
expr1 = x + 2 * y + 3 * z + 4.0
expr2 = x * x + 2 * x * y + 3 * z + 4.0
```

The first expression is linear, while the second is quadratic. An expression is typically then passed to setObjective (to set the optimization objective) or addConstr (to add a constraint).

Var.getAttr()

```
getAttr ( attrname )
```

Query the value of a variable attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(var.getAttr(GRB.Attr.X))
print(var.getAttr("x"))
```

Var.sameAs()

```
sameAs ( var2 )
```

Check whether two variable objects refer to the same variable.

Arguments:

`var2`: The other variable.

Return value:

Boolean result indicates whether the two variable objects refer to the same model variable.

Example usage:

```
print(model.getVars()[0].sameAs(model.getVars()[1]))
```

Var.index

`index`

This property returns the current index, or order, of the variable in the underlying constraint matrix.

Return value:

$= -2$: removed

$= -1$: not in model

≥ 0 : index of the variable in the model

Note that the index of a variable may change after subsequent model modifications.

Example usage:

```
v = model.getVars()[0]
print(v.index) # Index will be 0
```

Var.setAttr()

`setAttr ( attrname, newvalue )`

Set the value of a variable attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set.

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

Example usage:

```
var.setAttr(GRB.Attr.UB, 0.0)
var.setAttr("ub", 0.0)
```

6.4 MVar

Gurobi matrix variable object. An `MVar` is a NumPy ndarray of Gurobi variables. Variables are always associated with a particular model. You typically create these objects using Model.addMVar.

You generally use `MVar` objects to build matrix expressions, typically using overloaded operators. You can build linear matrix expressions or quadratic matrix expressions:

```
expr1 = A @ x
expr2 = A @ x + B @ y + z
expr3 = x @ A @ x + y @ B @ y
```

The first two expressions are linear, while the third is quadratic.

Dimensions and data types must always be compatible. In the examples above, matrix A must be either a NumPy ndarray with two dimensions or a SciPy sparse matrix (which will always have two dimensions), and x must be a 1-D `MVar`. In `expr1`, the size of the second dimension of A must be equal to the length of x. The same must be true of B and y in `expr2`. In addition, the size of the first dimension of A in `expr2` must be equal to the size of the first dimension of B, and also to the length of z.

For `expr3`, the size of the first dimension of A must be equal to the length of the `MVar` on the left, and the size of the second dimension must be equal to the length of the `MVar` on the right. The same is true for B.

An expression is typically then passed to setObjective (to set the optimization objective) or addConstr (to add a constraint).

Variable objects have a number of attributes. The full list can be found in the Attributes section of this document. Some variable attributes can only be queried, while others can also be set. Recall that the Gurobi optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about variable attributes. Consider the `lb` attribute. Its value can be queried using `var.lb`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `var.LB`. It can be set using a standard assignment statement (e.g., `var.lb = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `x` attribute), so attempts to assign new values to them will raise an exception.

You can also use MVar.getAttr/ MVar.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.LB`).

MVar()

```
MVar ( vars )
```

MVar constructor. Create an `MVar` object from a list of Var objects. Note that `MVar` objects are typically created using Model.addMVar.

Arguments:

`vars`: List of Gurobi `Var` objects.

Return value:

The new MVar object.

Example usage:

```
x = MVar(model.getVars())
```

MVar.copy()

`copy ( )`

Copy an `MVar` object.

Arguments:

Return value:

The new object.

Example usage:

```
print(copy = mvar.copy)
```

MVar.getAttr()

`getAttr ( attrname )`

Query the value of an attribute for a matrix variable. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

The result is returned in a NumPy ndarray with the same shape as the `MVar` object.

Arguments:

`attrname`: The attribute being queried.

Return value:

Current values for the requested attribute.

Example usage:

```
print(var.getAttr(GRB.Attr.X))
print(var.getAttr("x"))
```

MVar.setAttr()

`setAttr ( attrname, newvalue )`

Set the value of a matrix variable attribute.

Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set.

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute. The shape must be the same as that of the input argument. The one exception is a scalar argument, where the argument is automatically promoted to have the right shape.

Example usage:

```
var.setAttr("ub", np.full((5,), 0)
var.setAttr(GRB.Attr.UB, 0.0)
var.setAttr("ub", 0.0)
```

MVar.sum()

sum ( )

Sum the variables in an `MVar`. Returns an MLinExpr.

Return value:

The sum, as a linear matrix expression.

Example usage:

```
x.sum()
x[:,1].sum()
```

6.5 Constr

Gurobi constraint object. Constraints are always associated with a particular model. You create a constraint object by adding a constraint to a model (using Model.addConstr), rather than by using a `Constr` constructor.

Constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some constraint attributes can only be queried, while others can also be set. Recall that the Gurobi optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, or Model.write on the associated model.

We should point out a few things about constraint attributes. Consider the **`rhs`** attribute. Its value can be queried using `constr.rhs`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `constr.rhs`. It can be set using a standard assignment statement (e.g., `constr.rhs = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `Pi` attribute), so attempts to assign new values to them will raise an exception.

You can also use Constr.getAttr/ Constr.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.RHS`).

Constr.getAttr()

```
getAttr  ( attrname )
```

Query the value of a constraint attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(constr.getAttr(GRB.Attr.Slack))
print(constr.getAttr("slack"))
```

Constr.index

```
index
```

This property returns the current index, or order, of the constraint in the underlying constraint matrix.

Return value:

$= -2$: removed

$= -1$: not in model

≥ 0 : index of the constraint in the model

Note that the index of a constraint may change after subsequent model modifications.

Example usage:

```
c = model.getConstrs()[0]
print(c.index) # Index will be 0
```

Constr.sameAs()

```
sameAs ( constr2 )
```

Check whether two constraint objects refer to the same constraint.

Arguments:

`constr2`: The other constraint.

Return value:

Boolean result indicates whether the two constraint objects refer to the same model constraint.

Example usage:

```
print(model.getConstrs()[0].sameAs(model.getConstrs()[1]))
```

Constr.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of a constraint attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set.

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

Example usage:

```
constr.setAttr(GRB.Attr.RHS, 0.0)
constr.setAttr("rhs", 0.0)
```

6.6 QConstr

Gurobi quadratic constraint object. Quadratic constraints are always associated with a particular model. You create a quadratic constraint object by adding a quadratic constraint to a model (using Model.addQConstr), rather than by using a `QConstr` constructor.

Quadratic constraint objects have a number of attributes. The full list can be found in the Attributes section of this document. Some constraint attributes can only be queried, while others can also be set. Recall that the Gurobi optimizer employs a lazy update approach, so changes to attributes don't take effect until the next call to Model.update, Model.optimize, Model.write on the associated model.

We should point out a few things about quadratic constraint attributes. Consider the `qcrhs` attribute. Its value can be queried using `qconstr.qcrhs`. The Gurobi library ignores letter case in attribute names, so it can also be queried as `qconstr.QCRHS`. It can be set using a standard assignment statement (e.g., `qconstr.qcrhs = 0`). However, as mentioned earlier, attribute modification is done in a lazy fashion, so you won't see the effect of the change immediately. And some attributes can not be set (e.g., the `qcpi` attribute), so attempts to assign new values to them will raise an exception.

You can also use QConstr.getAttr/ QConstr.setAttr to access attributes. The attribute name can be passed to these routines as a string, or you can use the constants defined in the GRB.Attr class (e.g., `GRB.Attr.QCRHS`).

QConstr.getAttr()

```
getAttr  ( attrname )
```

Query the value of a quadratic constraint attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(qconstr.getAttr(GRB.Attr.QCSense))
print(qconstr.getAttr("qcsense"))
```

QConstr.setAttr()

```
setAttr  ( attrname, newvalue )
```

Set the value of a quadratic constraint attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set.

Arguments:

attrname: The attribute being modified.

newvalue: The desired new value of the attribute.

Example usage:

```
constr.setAttr(GRB.Attr.QCRHS, 0.0)
constr.setAttr("qcrhs", 0.0)
```

6.7 SOS

Gurobi SOS constraint object. SOS constraints are always associated with a particular model. You create an SOS object by adding an SOS constraint to a model (using Model.addSOS), rather than by using an SOS constructor. Similarly, SOS constraints are removed using the Model.remove method.

An SOS constraint can be of type 1 or 2 (`GRB.SOS_TYPE1` or `GRB.SOS_TYPE2`). A type 1 SOS constraint is a set of variables where at most one variable in the set may take a value other than zero. A type 2 SOS constraint is an ordered set of variables where at most two variables in the set may take non-zero values. If two take non-zero values, they must be contiguous in the ordered set.

SOS constraint objects have one attribute, `IISSOS`, which can be queried with the SOS.getAttr method.

SOS.getAttr()

```
getAttr ( attrname )
```

Query the value of an SOS attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(sos.getAttr(GRB.Attr.IISSOS))
```

6.8 GenConstr

Gurobi general constraint object. General constraints are always associated with a particular model. You add a general constraint to a model either by using one of the Model.addGenConstrXxx method, or by using Model.addConstr or Model.addConstrs plus a general constraint helper function).

General constraint objects have a number of attributes, which can be queried with the GenConstr.getAttr method. The full list can be found in the Attributes section of this document.

GenConstr.getAttr()

```
getAttr ( attrname )
```

Query the value of a general constraint attribute. The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the requested attribute doesn't exist or can't be queried.

Arguments:

`attrname`: The attribute being queried.

Return value:

The current value of the requested attribute.

Example usage:

```
print(genconstr.getAttr(GRB.Attr.GenConstrType))
print(genconstr.getAttr("GenConstrType"))
```

GenConstr.setAttr()

```
setAttr ( attrname, newvalue )
```

Set the value of a general constraint attribute. Note that, due to our lazy update approach, the change won't actually take effect until you update the model (using Model.update), optimize the model (using Model.optimize), or write the model to disk (using Model.write).

The full list of available attributes can be found in the Attributes section.

Raises an `AttributeError` if the specified attribute doesn't exist or can't be set.

Arguments:

`attrname`: The attribute being modified.

`newvalue`: The desired new value of the attribute.

6.9 LinExpr

Gurobi linear expression object. A linear expression consists of a constant term, plus a list of coefficient-variable pairs that capture the linear terms. Linear expressions are used to build linear objective and constraints. They are temporary objects that typically have short lifespans.

You generally build linear expressions using overloaded operators. For example, if `x` is a Var object, then `x + 1` is a LinExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x + 2 * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x`, or `expr2 -= expr1`).

The full list of overloaded operators on LinExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, and `/`. In Python parlance, we've defined the following `LinExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, `__imul__`, and `__div__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from linear expressions.

You can also use add or addTerms to modify expressions. The LinExpr() constructor can be used to build expressions. Another option is quicksum; it is a more efficient version of the Python `sum` function. Terms can be removed from an expression using remove.

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- While the Python `sum` function can be used to build expressions, it should be avoided. Its cost is quadratic in the length of the expression.
- For similar reasons, you should avoid using `expr = expr + x` in a loop. Building large expressions in this way also leads to quadratic runtimes.
- The quicksum function is much quicker than `sum`, as are loops over `expr += x` or expr.add(x). These approaches are fast enough for most programs, but they may still be expensive for very large expressions.
- The two most efficient ways to build large linear expressions are addTerms or the LinExpr() constructor.

Individual terms in a linear expression can be queried using the getVar, getCoeff, and getConstant methods. You can query the number of terms in the expression using the size method.

Note that a linear expression may contain multiple terms that involve the same variable. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar).

LinExpr()

```
LinExpr ( arg1=0.0, arg2=None )
```

Linear expression constructor. Note that you should generally use overloaded operators instead of the explicit constructor to build linear expression objects.

This constructor takes multiple forms. You can initialize a linear expression using a constant (`LinExpr(2.0)`), a variable (`LinExpr(x)`), an expression (`LinExpr(2*x)`), a pair of lists containing coefficients and variables, respectively (`LinExpr([1.0, 2.0], [x, y])`), or a list of coefficient-variable tuples (`LinExpr([(1.0, x), (2.0, y), (1.0, z)])`).

Return value:

A linear expression object.

Example usage:

```
expr = LinExpr(2.0)
expr = LinExpr(2*x)
expr = LinExpr([1.0, 2.0], [x, y])
expr = LinExpr([(1.0, x), (2.0, y), (1.0, z)])
```

LinExpr.add()

```
add ( expr, mult=1.0 )
```

Add one linear expression into another. Upon completion, the invoking linear expression will be equal to the sum of itself and the argument expression.

Arguments:

`expr`: Linear expression to add.

`mult (optional)`: Multiplier for argument expression.

Example usage:

```
e1 = x + y
e1.add(z, 3.0)
```

LinExpr.addConstant()

```
addConstant ( c )
```

Add a constant into a linear expression.

Arguments:

`c`: Constant to add to expression.

Example usage:

```
expr = x + 2 * y
expr.addConstant(0.1)
```

LinExpr.addTerms()

```
addTerms ( coeffs, vars )
```

Add new terms into a linear expression.

Arguments:

`coeffs`: Coefficients for new terms; either a list of coefficients or a single coefficient. The two arguments must have the same size.

`vars`: Variables for new terms; either a list of variables or a single variable. The two arguments must have the same size.

Example usage:

```
expr.addTerms(1.0, x)
expr.addTerms([2.0, 3.0], [y, z])
```

LinExpr.clear()

```
clear ( )
```

Set a linear expression to 0.

Example usage:

```
expr.clear()
```

LinExpr.copy()

```
copy ( )
```

Copy a linear expression

Return value:

Copy of input expression.

Example usage:

```
e0 = 2 * x + 3
e1 = e0.copy()
```

LinExpr.getConstant()

```
getConstant ( )
```

Retrieve the constant term from a linear expression.

Return value:

Constant from expression.

Example usage:

```
e = 2 * x + 3
print(e.getConstant())
```

LinExpr.getCoeff()

```
getCoeff ( i )
```

Retrieve the coefficient from a single term of the expression.

Return value:

Coefficient for the term at index `i` in the expression.

Example usage:

```
e = x + 2 * y + 3
print(e.getCoeff(1))
```

LinExpr.getValue()

```
getValue ( )
```

Compute the value of an expression using the current solution.

Return value:

The value of the expression.

Example usage:

```
obj = model.getObjective()
print(obj.getValue())
```

LinExpr.getVar()

`getVar ( i )`

Retrieve the variable object from a single term of the expression.

Return value:

Variable for the term at index `i` in the expression.

Example usage:

```
e = x + 2 * y + 3
print(e.getVar(1))
```

LinExpr.remove()

`remove ( item )`

Remove a term from a linear expression.

Arguments:

`item`: If `item` is an integer, then the term stored at index `item` of the expression is removed. If `item` is a Var, then all terms that involve `item` are removed.

Example usage:

```
e = x + 2 * y + 3
e.remove(x)
```

LinExpr.size()

`size ( )`

Retrieve the number of terms in the linear expression (not including the constant).

Return value:

Number of terms in the expression.

Example usage:

```
e = x + 2 * y + 3
print(e.size())
```

LinExpr.__eq__()

`__eq__ ( )`

Overloads the == operator, creating a TempConstr object that captures an equality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x + y == 1)
```

LinExpr.__le__()

`__le__ ( )`

Overloads the `<=` operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x + y <= 1)
```

LinExpr.__ge__()

`__ge__ ( arg )`

Overloads the `>=` operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x + y >= 1)
```

6.10 QuadExpr

Gurobi quadratic expression object. A quadratic expression consists of a linear expression plus a list of coefficient-variable-variable triples that capture the quadratic terms. Quadratic expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

You generally build quadratic expressions using overloaded operators. For example, if `x` is a Var object, then `x * x` is a QuadExpr object. Expressions can be built from constants (e.g., `expr = 0`), variables (e.g., `expr = 1 * x *x + 2 * x * y`), or from other expressions (e.g., `expr2 = 2 * expr1 + x * x`, or `expr3 = expr1 + 2 * expr2`). You can also modify existing expressions (e.g., `expr += x * x`, or `expr2 -= expr1`).

The full list of overloaded operators on QuadExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, and `/`. In Python parlance, we've defined the following `QuadExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, `__imul__`, and `__div__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from quadratic expressions.

You can use quicksum to build quadratic expressions; it is a more efficient version of the Python `sum` function. You can also use add or addTerms to modify expressions. Terms can be removed from an expression using remove.

Note that the cost of building expressions depends heavily on the approach you use. While you can generally ignore this issue when building small expressions, you should be aware of a few efficiency issues when building large expressions:

- While the Python `sum` function can be used to build expressions, it should be avoided. Its cost is quadratic in the length of the expression.
- For similar reasons, you should avoid using `expr = expr + x*x` in a loop. Building large expressions in this way also leads to quadratic runtimes.
- The quicksum function is much quicker than `sum`, as are loops over `expr += x*x` or expr.add(x*x). These approaches are fast enough for most programs, but they may still be expensive for very large expressions.
- The most efficient way to build a large quadratic expression is with a single call to addTerms.

Individual quadratic terms in a quadratic expression can be queried using the getVar1, getVar2, and getCoeff methods. You can query the number of quadratic terms in the expression using the size method. To query the constant and linear terms associated with a quadratic expression, use getLinExpr to obtain the linear portion of the quadratic expression, and then use the getVar, getCoeff, and getConstant methods on this LinExpr object. Note that a quadratic expression may contain multiple terms that involve the same variable pair. These duplicate terms are merged when creating a constraint from an expression, but they may be visible when inspecting individual terms in the expression (e.g., when using getVar1 and getVar2).

QuadExpr()

```
QuadExpr  ( expr = None )
```

Quadratic expression constructor. Note that you should generally use overloaded operators instead of the explicit constructor to build quadratic expression objects.

Arguments:

`expr (optional)`: Initial value of quadratic expression. Can be a `LinExpr` or a `QuadExpr`. If no argument is specified, the initial expression value is 0.

Return value:

A quadratic expression object.

Example usage:

```
expr = QuadExpr()
expr = QuadExpr(2*x)
expr = QuadExpr(x*x + y+y)
```

QuadExpr.add()

```
add ( expr, mult=1.0 )
```

Add an expression into a quadratic expression. Argument can be either a linear or a quadratic expression. Upon completion, the invoking quadratic expression will be equal to the sum of itself and the argument expression.

Arguments:

`expr`: Linear or quadratic expression to add.

`mult (optional)`: Multiplier for argument expression.

Example usage:

```
expr = x * x + 2 * y * y
expr.add(z * z, 3.0)
```

QuadExpr.addConstant()

```
addConstant ( c )
```

Add a constant into a quadratic expression.

Arguments:

`c`: Constant to add to expression.

Example usage:

```
expr = x * x + 2 * y * y + z
expr.addConstant(0.1)
```

QuadExpr.addTerms()

```
addTerms ( coeffs, vars, vars2=None )
```

Add new linear or quadratic terms into a quadratic expression.

Arguments:

`coeffs`: Coefficients for new terms; either a list of coefficients or a single coefficient. The arguments must have the same size.

vars: Variables for new terms; either a list of variables or a single variable. The arguments must have the same size.

vars2 (optional): Variables for new quadratic terms; either a list of variables or a single variable. Only present when you are adding quadratic terms. The arguments must have the same size.

Example usage:

```
expr.addTerms(1.0, x)
expr.addTerms([2.0, 3.0], [y, z])
expr.addTerms([2.0, 3.0], [x, y], [y, z])
```

QuadExpr.clear()

```
clear ( )
```

Set a quadratic expression to 0.

Example usage:

```
expr.clear()
```

QuadExpr.copy()

```
copy ( )
```

Copy a quadratic expression

Return value:

Copy of input expression.

Example usage:

```
e0 = x * x + 2 * y * y + z
e1 = e0.copy()
```

QuadExpr.getCoeff()

```
getCoeff ( i )
```

Retrieve the coefficient from a single term of the expression.

Return value:

Coefficient for the quadratic term at index `i` in the expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.getCoeff(1))
```

QuadExpr.getLinExpr()

```
getLinExpr ( )
```

A quadratic expression is represented as a linear expression, plus a list of quadratic terms. This method retrieves the linear expression associated with the quadratic expression.

Return value:

Linear expression from quadratic expression.

Example usage:

```
expr = x * x + 2 * y * y + z
le = expr.getLinExpr()
```

QuadExpr.getValue()

```
getValue ( )
```

Compute the value of an expression using the current solution.

Return value:

The value of the expression.

Example usage:

```
obj = model.getObjective()
print(obj.getValue())
```

QuadExpr.getVar1()

```
getVar1 ( i )
```

Retrieve the first variable for a single quadratic term of the quadratic expression.

Return value:

First variable associated with the quadratic term at index `i` in the quadratic expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.getVar1(1))
```

QuadExpr.getVar2()

```
getVar2 ( i )
```

Retrieve the second variable for a single quadratic term of the quadratic expression.

Return value:

Second variable associated with the quadratic term at index `i` in the quadratic expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.getVar2(1))
```

QuadExpr.remove()

```
remove ( item )
```

Remove a term from a quadratic expression.

Arguments:

`item`: If `item` is an integer, then the quadratic term stored at index `item` of the expression is removed. If `item` is a Var, then all quadratic terms that involve `item` are removed.

Example usage:

```
expr = x * x + 2 * y * y + z
expr.remove(x)
```

QuadExpr.size()

| **size** ( )

Retrieve the number of quadratic terms in the expression.

Return value:

Number of quadratic terms in the expression.

Example usage:

```
expr = x * x + 2 * y * y + z
print(expr.size())
```

QuadExpr.__eq__()

| **__eq__** ( )

Overloads the == operator, creating a TempConstr object that captures an equality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x*x + y*y == 1)
```

QuadExpr.__le__()

| **__le__** ( )

Overloads the <= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x*x + y*y <= 1)
```

QuadExpr.__ge__()

| **__ge__** (arg)

Overloads the >= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x*x + y*y >= 1)
```

6.11 GenExpr

Gurobi general expression object. Objects of this class are created by a set of general constraint helper functions functions. They are temporary objects, meant to be used in conjunction with overloaded operators to build TempConstr objects, which are then passed to addConstr or addConstrs to build general constraints.

To be more specific, the following creates a `GenExpr` object...

```
max_(x, y)
```

The following creates a `TempConstr` object...

```
z == max_(x, y)
```

The following adds a general constraint to a model...

```
model.addConstr(z == max_(x, y))
```

Please refer to the TempConstr documentation for more information on building general constraints.

6.12 MLinExpr

Gurobi linear matrix expression object. A linear matrix expression consists of a matrix-vector product, where the matrix is a NumPy dense matrix or a SciPy sparse matrix and the vector is a Gurobi MVar object, plus an optional constant vector of compatible dimensions (i.e., $Ax + b$). Linear matrix expressions are used to build linear objectives and constraints. They are temporary objects that typically have short lifespans.

You generally build linear matrix expressions using overloaded operators, typically by multiplying a 2-D matrix (dense or sparse) by a 1-D MVar object using the Python matrix multiply (`@`) operator (e.g., `expr = A @ x`). You can also promote an `MVar` object to an `MLinExpr` using arithmetic expressions (e.g., `expr = x + 1`). Most arithmetic operations are supported on `MLinExpr` objects, including addition and subtraction (e.g., `expr = A @ x - B @ y`), and multiplication by a constant (e.g. `expr = 2 * A @ x`).

An `MLinExpr` object has a `shape`, defined similarly to that of other NumPy ndarray objects. Due to the way it is defined, the shape will always be 1-dimensional, with the length reflecting the size of the corresponding matrix-vector result. To give an example, forming `A @ x` where `A` has shape `(10,3)` and `x` has shape `(3,)` gives a result with shape `(10,)`.

When working with `MLinExpr` objects, you of course need to make sure that the shapes are compatible. If you want to form `A @ x`, then `A` must be a 2-D array, `x` must be a 1-D array, and the size of the second dimension of `A` must be equal to the size of `x`. Similarly, adding an object into an `MLinExpr` object (including another `MLinExpr`) requires an object of the same shape. The one exception is a constant, which is automatically promoted to have a compatible shape.

The full list of overloaded operators on MLinExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, and `@`. In Python parlance, we've defined the following `MLinExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, `__imul__`, `__matmul__`, and `__rmatmul__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from linear matrix expressions.

Note that the Python matrix multiplication operator (`@`) was introduced in Python version 3.5; it isn't available from Python 2.7.

MLinExpr.copy()

```
copy ( )
```

Create a (shallow) copy of a linear matrix expression.

Return value:

Copy of expression object.

Example usage:

```
expr2 = expr1.copy()
```

MLinExpr.getValue()

```
getValue ( )
```

Compute the value of a linear matrix expression using the current solution.

Return value:

Value of expression (1-D array).

Example usage:

```
expr = A @ x + b
expr.getValue()
```

MLinExpr.__eq__()

```
__eq__ ( )
```

Overloads the == operator, creating a TempConstr object that captures a 1-D array of equality constraints. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(A @ x == 1)
```

MLinExpr.__le__()

```
__le__ ( )
```

Overloads the <= operator, creating a TempConstr object that captures a 1-D array of inequality constraints. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(A @ x <= 1)
```

MLinExpr.__ge__()

```
__ge__ ( arg )
```

Overloads the >= operator, creating a TempConstr object that captures a 1-D array of inequality constraints. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(A @ x >= 1)
```

6.13 MQuadExpr

Gurobi quadratic matrix expression object. Quadratic matrix expressions are used to build quadratic objective functions and quadratic constraints. They are temporary objects that typically have short lifespans.

You generally build quadratic matrix expressions using overloaded operators. For example, if `x` is an MVar object and `A` is a 2-D matrix (dense or sparse), then `x @ A @ x` and `x @ x` are both MQuadExpr objects. Most arithmetic operations are support on `MQuadExpr` objects, including addition and subtraction (e.g., `expr = x @ A @ x - y @ B @ y`), and multiplication by a constant (e.g. `expr = 2 * x @ A @ y`).

The full list of overloaded operators on MQuadExpr objects is as follows: `+`, `+=`, `-`, `-=`, `*`, `*=`, and `/`. In Python parlance, we've defined the following `QuadExpr` functions: `__add__`, `__radd__`, `__iadd__`, `__sub__`, `__rsub__`, `__isub__`, `__neg__`, `__mul__`, `__rmul__`, and `__imul__`.

We've also overloaded the comparison operators (==, <=, and >=), to make it easier to build constraints from quadratic expressions.

Note that the Python matrix multiplication operator (`@`) was introduced in Python version 3.5; it isn't available from Python 2.7.

Note that a quadratic matrix expression always produces a scalar result (a result with shape `(1,)`). You can add linear terms into a quadratic matrix expression, but for the dimensions to be compatible they must also have shape `(1,)`.

MQuadExpr.copy()

```
copy ( )
```

Create a (shallow) copy of a quadratic matrix expression.

Return value:

Copy of expression object.

Example usage:

```
expr2 = expr1.copy()
```

MQuadExpr.getValue()

```
getValue ( )
```

Compute the value of a quadratic matrix expression using the current solution.

Return value:

Value of expression (scalar).

Example usage:

```
qexpr = x @ A @ x
qexpr.getValue()
```

MQuadExpr.__eq__()

```
__eq__ ( )
```

Overloads the == operator, creating a TempConstr object that captures an equality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x @ Q @ y == 1)
```

MQuadExpr.__le__()

```
__le__ ( )
```

Overloads the <= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x @ Q @ y <= 1)
```

MQuadExpr.__ge__()

```
__ge__ ( arg )
```

Overloads the >= operator, creating a TempConstr object that captures an inequality constraint. The result is typically immediately passed to Model.addConstr.

Return value:

A `TempConstr` object.

Example usage:

```
m.addConstr(x @ Q @ y >= 1)
```

6.14 TempConstr

Gurobi temporary constraint object. Objects of this class are created as intermediate results when building constraints using overloaded operators. There are no member functions on this class. Instead, `TempConstr` objects are created by a set of functions on Var, MVar, LinExpr, QuadExpr, MLinExpr, MQuadExpr, and GenExpr objects (e.g., ==, <=, and >=). You will generally never store objects of this class in your own variables.

The `TempConstr` object allows you to create several different types of constraints:

- **Linear Constraint:** an expression of the form `Expr1 sense Expr1`, where `Expr1` and `Expr2` are LinExpr objects, Var objects, or constants, and `sense` is one of `==`, `<=` or `>=`. For example, `x + y <= 1 + z` is a linear constraint, as is `x + y == 5`. Note that `Expr1` and `Expr2` can't both be constants.

- **Ranged Linear Constraint:** an expression of the form `LinExpr == [Const1, Const2]`, where `Const1` and `Const2` are constants and `LinExpr` is a LinExpr object. For example, `x + y == [1, 2]` is a ranged linear constraint.

- **Quadratic Constraint:** an expression of the form `Expr1 sense Expr2`, where `Expr1` and `Expr2` are QuadExpr objects, LinExpr objects, Var objects, or constants, and `sense` is one of `==`, `<=` or `>=`. For example, `x*x + y*y <= 3` is a quadratic constraint, as is `x*x + y*y <= z*z`. Note that one of `Expr1` or `Expr2` must be a QuadExpr (otherwise, the constraint would be linear).

- **Linear Matrix Constraint:** an expression of the form `Expr1 sense Expr1`, where one or both of `Expr1` and `Expr2` are MLinExpr objects and `sense` is one of `==`, `<=` or `>=`. For example, `A @ x <= 1` is a linear matrix constraint, as is `A @ x == B @ y`.

- **Quadratic Matrix Constraint:** an expression of the form `Expr1 sense Expr2`, where one or both of `Expr1` and `Expr2` are MQuadExpr objects and `sense` is one of `==`, `<=` or `>=`. For example, `x @ Q @ y <= 3` is a quadratic constraint, as is `x @ Q @ x <= y @ A @ y`.

- **Absolute Value Constraint:** an expression of the form `x == abs_(y)`, where `x` and `y` must be Var objects.

- **Logical Constraint:** an expression of the form `x == op_(y)`, where `x` is a binary Var object, and `y` is a binary Var, a list of binary Var, or a tupledict of binary Var, and `op_` is either `and_` or `or_` (or the Python-specific variants, `all_` and `any_`).

- **Min or Max Constraint:** an expression of the form `x == op_(y)`, where `x` is a Var object, and `y` is a Var, a list of Var and constants, or a tupledict of Var, and `op_` is one of `min_` or `max_`.

- **Indicator Constraint:** an expression of the form `(x == b) >> (Expr1 sense Expr2)`, where `x` is a binary Var object, `b` is either `0` or `1`; `Expr1` and `Expr2` are LinExpr objects, Var objects, or constants, and `sense` is one of `==`, `<=` or `>=`. Parenthesizing both expressions is required. For example, `(x == 1) >> (y + w <= 5)` is an indicator constraint, indicating that whenever the binary variable `x` takes the value `1` then the linear constraint `y + w <= 5` must hold.

Consider the following examples:

```
model.addConstr(x + y == 1);
model.addConstr(x + y == [1, 2]);
model.addConstr(x*x + y*y <= 1);
model.addConstr(A @ x <= 1);
model.addConstr(x @ A @ x <= 1);
model.addConstr(x == abs_(y));
model.addConstr(x == or_(y, z));
model.addConstr(x == max_(y, z));
model.addConstr((x == 1) >> (y + z <= 5));
```

In each case, the overloaded comparison operator creates an object of type `TempConstr`, which is then immediately passed to method Model.addConstr.

6.15 Column

Gurobi column object. A column consists of a list of coefficient, constraint pairs. Columns are used to represent the set of constraints in which a variable participates, and the associated coefficients. They are temporary objects that typically have short lifespans.

You generally build columns using the Column constructor. Terms can be added to an existing column using addTerms. Terms can also be removed from a column using remove.

Individual terms in a column can be queried using the getConstr, and getCoeff methods. You can query the number of terms in the column using the size method.

Column()

```
Column ( coeffs=None, constrs=None )
```

Column constructor.

Arguments:

`coeffs (optional)`: Lists the coefficients associated with the members of `constrs`.

`constrs (optional)`: Constraint or constraints that participate in expression. If `constrs` is a list, then `coeffs` must contain a list of the same length. If `constrs` is a single constraint, then `coeffs` must be a scalar.

Return value:

An expression object.

Example usage:

```
col = Column()
col = Column(3, c1)
col = Column([1.0, 2.0], [c1, c2])
```

Column.addTerms()

```
addTerms ( coeffs, constrs )
```

Add new terms into a column.

Arguments:

`coeffs`: Coefficients for added constraints; either a list of coefficients or a single coefficient. The two arguments must have the same size.

`constrs`: Constraints to add to column; either a list of constraints or a single constraint. The two arguments must have the same size.

Example usage:

```
col.addTerms(1.0, x)
col.addTerms([2.0, 3.0], [y, z])
```

Column.clear()

```
clear ( )
```

Remove all terms from a column.

Example usage:

```
col.clear()
```

Column.copy()

`copy ( )`

Copy a column.

Return value:

Copy of input column.

Example usage:

```
col0 = Column(1.0, c0)
col1 = col0.copy()
```

Column.getCoeff()

`getCoeff ( i )`

Retrieve the coefficient from a single term in the column.

Return value:

Coefficient for the term at index `i` in the column.

Example usage:

```
col = Column([1.0, 2.0], [c0, c1])
print(col.getCoeff(1))
```

Column.getConstr()

`getConstr ( i )`

Retrieve the constraint object from a single term in the column.

Return value:

Constraint for the term at index `i` in the column.

Example usage:

```
col = Column([1.0, 2.0], [c0, c1])
print(col.getConstr(1))
```

Column.remove()

`remove ( item )`

Remove a term from a column.

Arguments:

`item`: If `item` is an integer, then the term stored at index `item` of the column is removed. If `item` is a Constr, then all terms that involve `item` are removed.

Example usage:

```
col = Column([1.0, 2.0], [c0, c1])
col.remove(c0)
```

Column.size()

```
size ( )
```

Retrieve the number of terms in the column.

Return value:

Number of terms in the column.

Example usage:

```
print(Column([1.0, 2.0], [c0, c1]).size())
```

6.16 Callbacks

Gurobi callback class. A callback is a user function that is called periodically by the Gurobi optimizer in order to allow the user to query or modify the state of the optimization. More precisely, if you pass a function that takes two arguments (`model` and `where`) as the argument to Model.optimize, your function will be called during the optimization. Your callback function can then call Model.cbGet to query the optimizer for details on the state of the optimization.

Gurobi callbacks can be used both to monitor the progress of the optimization and to modify the behavior of the Gurobi optimizer. A simple user callback function might call Model.cbGet to produce a custom display, or perhaps to terminate optimization early (using Model.terminate). More sophisticated MIP callbacks might use Model.cbGetNodeRel or Model.cbGetSolution to retrieve values from the solution to the current node, and then use Model.cbCut or Model.cbLazy to add a constraint to cut off that solution, or Model.cbSetSolution to import a heuristic solution built from that solution. For multi-objective problems, you might use Model.cbStopOneMultiObj to interrupt the optimization process of one of the optimization steps in a multi-objective MIP problem without stopping the hierarchical optimization process.

The Gurobi callback class provides a set of constants that are used within the user callback function. The first set of constants in this class list the options for the `where` argument to the user callback function. The `where` argument indicates from where in the optimization process the user callback is being called. Options are listed in the Callback Codes section of this document.

The other set of constants in this class list the options for the `what` argument to Model.cbGet. The `what` argument is used by the user callback to indicate what piece of status information it would like to retrieve. The full list of options can be found in the Callback Codes section. As with the `where` argument, you refer to a `what` constant through `GRB.Callback`. For example, the simplex objective value would be requested using `GRB.Callback.SPX_OBJVAL`.

If you would like to pass data to your callback function, you can do so through the Model object. For example, if your program includes the statement `model._value = 1` before the optimization begins, then your callback function can query the value of `model._value`. Note that the name of the user data field must begin with an underscore.

When solving a model using multiple threads, the user callback is only ever called from a single thread, so you don't need to worry about the thread-safety of your callback.

Note that changing parameters from within a callback is not supported, doing so may lead to undefined behavior.

You can look at `callback.py` in the examples directory for details of how to use Gurobi callbacks.

6.17 GurobiError

Gurobi exception object. Upon catching an exception `e`, you can examine `e.errno` (an integer) or `e.message` (a string). A list of possible values for `errno` can be found in the Error Code section. `message` provides additional information on the source of the error.

6.18 Env

Gurobi environment object. Note that environments play a much smaller role in the Python interface than they do in other Gurobi language APIs, mainly because the Python interface has a default environment. Unless you explicitly pass your own environment to routines that require an environment, the default environment will be used.

The primary situations where you will want to use your own environment are:

- When you are using a Gurobi Compute Server and want to choose the server from within your program.
- When you need control over garbage collection of your environment. The Gurobi Python interface maintains a reference to the default environment, so by default it will never be garbage collected. By creating your own environment, you can control exactly when your program releases any licensing tokens or Compute Servers it is using.
- When you are using concurrent environments in one of the concurrent optimizers.

It is good practice to use the `with` keyword when dealing with environment (and model) objects. That way the resources tied to these objects are properly released even if an exception is raised at some point. The following example illustrates two typical use patterns.

Example usage:

```
import gurobipy as gp
with gp.Env("gurobi.log") as env, gp.Model(env=env) as model:
    # Populate model object here...
    m.optimize()

with gp.Env(empty=True) as env:
    env.setParam("ComputeServer", "myserver1:32123")
    env.setParam("ServerPassword", "pass")
    env.start()
    with gp.Model(env=env) as model:
        # Populate model object here...
        model.optimize()
```

Note that you can manually remove the reference to the default environment by calling disposeDefaultEnv. After calling this, and after all models built within the default environment are garbage collected, the default environment will be garbage collected as well. A new default environment will be created automatically if you call a routine that needs one.

Env()

```
Env ( logfilename="", empty=False, params=None )
```

Env constructor. You will generally want to use the default environment in Gurobi Python programs. The exception is when you want explicit control over environment garbage collection. By creating your own environment object and always passing it to methods that take an environment as input (read or the Model constructor), you will avoid creating the default environment. Once every model created using an Env object is garbage collected, and once the Env object itself is no longer referenced, the garbage collector will reclaim the environment and release all associated resources.

If the environment is not empty, This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`logfilename`: Name of the log file for this environment. Pass an empty string if you don't want a log file.

`empty`: Indicates whether the environment should be empty. You should use `empty=True` if you want to set parameters before actually starting the environment. This can be useful if you want to connect to a Compute Server, a Token Server, the Gurobi Instant Cloud or a Cluster Manager. See the Environment Section for more details.

`params`: A dict containing Gurobi parameter/value pairs that should be set already upon environment creation. Any server related parameters can be set through this dict, too.

Return value:

New environment object.

Example usage:

```
env = Env("gurobi.log")
m = read("misc07.mps", env)
m.optimize()
```

Example usage:

```
p = {"ComputeServer": "localhost:33322",
     "ServerPassword": "pass",
     "TimeLimit": 120.0}
with Env(params=p) as env, read('misc07.mps', env=env) as model:
    model.optimize()
```

Env.ClientEnv()

```
Env.ClientEnv  ( logfilename="", computeServer="", router="", password="",
                 group="", CStlsInsecure=0, priority=0, timeout=-1 )
```

Compute Server Env constructor. Creates a client environment on a compute server. If all compute servers are at capacity, this command will cause a job to be placed in the compute server queue, and the command will return an environment once capacity is available.

Client environments have limited uses in the Python environment. You can use a client environment as an argument to the Model constructor, to indicate that a model should be constructed on a Compute Server, or as an argument to the global read function, to indicate that the result of reading the file should be place on a Compute Server.

This method will also check the current working directory for a file named **`gurobi.env`**, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Arguments:

`logfilename`: Name of the log file for this environment. Pass an empty string if you don't want a log file.

`computeServer`: A Compute Server. You can refer to the server using its name or its IP address. If you are using a non-default port, the server name should be followed by the port number (e.g., `server1:61000`)

`router`: The router for a Compute Server cluster. A router can be used to improve the robustness of a Compute Server deployment. You should refer to the router using either its name or its IP address. If no router is used (which is the typical case), pass an empty string.

`password`: The password for gaining access to the specified Compute Server cluster. Pass an empty string if no password is required.

`group`: The name of the Compute Server group.

`CStlsInsecure`: Indicates whether to use insecure mode in the TLS (Transport Layer Security). Set this to 0 unless your server administrator tells you otherwise.

`priority`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs. Depending on the configuration of the server, a job with priority 100 runs immediately, bypassing the job queue and ignoring the job limit on the server. You should exercise caution with priority 100 jobs, since they can severely overload a server, which can cause jobs to fail, and in extreme cases can cause the server to crash. This behavior is managed by the `HARDJOBLIMIT`, and is disabled by default. Refer to the Gurobi Remote Services Reference Manual for more information on starting Compute Server options.

`timeout`: Queue timeout (in seconds). If the job doesn't reach the front of the queue before the specified timeout, the call will exit with a `JOB_REJECTED` error. Use -1 to indicate that the call should never timeout.

Return value:

New environment object.

Example usage:

```
env = Env.ClientEnv("client.log", "server1.mycompany.com,server2.mycompany.com")
m = read("misc07.mps", env)
m.optimize()
```

Env.CloudEnv()

```
Env.CloudEnv ( logfilename="", accessID, secretKey, pool="", priority=0 )
```

Instant Cloud Env constructor. Creates a Gurobi environment on an Instant Cloud server. Uses an existing Instant Cloud machine if one is currently active within the specified machine pool, and launches a new one otherwise. Note that launching a machine can take a few minutes.

Once an Instant Cloud server is active (either because it was already active or because the launch of a new server completed), this command places a job in the server queue. If the server has sufficient capacity, the job will start immediately. Otherwise, it is placed in the server queue and this command returns once capacity becomes available.

You should visit the Gurobi Instant Cloud site to obtain your `accessID` and `secretKey`, configure your machine pools, and perform other cloud setup and maintenance tasks.

Note that you should keep your secretKey private. Sharing it with others will allow them to launch Instant Cloud instances in your account.

This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter).

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Most methods in the Gurobi Python interface will use the default environment, so you'll need to take special action to use the cloud environment created by this method. You can use a cloud environment as an argument to the Model constructor, to indicate that a model should be constructed on the given Instant Cloud server, or as an argument to the global read function, to indicate that the result of reading the file should be placed on the given Instant Cloud Server.

Arguments:

`logfilename`: The name of the log file for this environment. May be `NULL` (or an empty string), in which case no log file is created.

`accessID`: The access ID for your Gurobi Instant Cloud license. This can be retrieved from the Gurobi Instant Cloud website. When used in combination with your `secretKey`, this allows you to launch Instant Cloud instances and submit jobs to them.

`secretKey`: The secret key for your Gurobi Instant Cloud license. This can be retrieved from the Gurobi Instant Cloud website. When used in combination with your `accessID`, this allows you to launch Instant Cloud instances and submit jobs to them. Note that you should keep your secret key private.

`pool`: The machine pool. Machine pools allow you to create fixed configurations on the Instant Cloud website (capturing things like type of machine, geographic region, etc.), and then launch and share machines from client programs without having to restart the configuration information each time you launch a machine. May be `NULL` (or an empty string), in which case your job will be launched in the default pool associated with your cloud license.

`priority`: The priority of the job. Priorities must be between -100 and 100, with a default value of 0 (by convention). Higher priority jobs are chosen from the server job queue before lower priority jobs.

Return value:

New environment object.

Example usage:

```
env = Env.CloudEnv("cloud.log",
                    "3d1ecef9-dfad-eff4-b3fa", "ae6L23alJe3+fas");
m = read("misc07.mps", env)
m.optimize()
```

Env.resetParams()

```
resetParams ( )
```

Reset the values of all parameters to their default values.

Example usage:

```
env.resetParams()
```

Env.setParam()

```
setParam ( paramname, newvalue )
```

Set the value of a parameter to a new value.

Arguments:

`paramname`: String containing the name of parameter that you would like to modify. The name can include '*' and '?' wildcards. If more than one parameter matches, the matching names are listed and none are modified. Note that case is ignored.

`newvalue`: Desired new value for parameter. Can be 'default', which indicates that the parameter should be reset to its default value.

Example usage:

```
env.setParam("Cuts", 2)
env.setParam("Heu*", 0.5)
env.setParam("*Interval", 10)
```

Env.start()

```
start ( )
```

Start an empty environment. If the environment has already been started, this method will do nothing. If the call fails, the environment will have the same state as it had before the call to this method.

This method will also populate any parameter (`ComputeServer`, `TokenServer`, `ServerPassword`, etc.) specified in your `gurobi.lic` file. This method will also check the current working directory for a file named `gurobi.env`, and it will attempt to read parameter settings from this file if it exists. The file should be in PRM format (briefly, each line should contain a parameter name, followed by the desired value for that parameter). After that, it will apply all parameter changes specified by the user prior to this call. Note that this might overwrite parameters set in the license file, or in the `gurobi.env` file, if present.

After all these changes are performed, the code will actually activate the environment, and make it ready to work with models.

In general, you should aim to create a single Gurobi environment in your program, even if you plan to work with multiple models. Reusing one environment is much more efficient than creating and destroying multiple environments. The one exception is if you are writing a multi-threaded program, since environments are not thread safe. In this case, you will need a separate environment for each of your threads.

Example usage:

```
env = Env(empty=True)
env.setParam('ComputeServer', 'server.mydomain.com:61000')
env.setParam('ServerPassword', 'mypassword')
env.start()
```

Env.writeParams()

```
writeParams ( filename )
```

Write all modified parameters to a file. The file is written in PRM format.

Example usage:

```
env.setParam("Heu*", 0.5)
env.writeParams("params.prm")  # file will contain changed parameter
system("cat params.prm")
```

Env.dispose()

```
dispose ( force=False )
```

Free all resources associated with this Env object.

Dispose of all models created in this environment before disposing of this Env object. An error will be raised if this is not the case. Use the 'force' parameter to override this behavior, and to proceed with the disposal nevertheless. When doing so, any resource (memory, etc.), consumed by still existing models cannot be reclaimed.

After this method is called, this Env object must no longer be used.

Example usage:

```
env = Env()
model = read("misc07.mps", env)
model.optimize()
model.dispose()
env.dispose()
```

6.19 Batch

Gurobi batch object. Batch optimization is a feature available with the Gurobi Cluster Manager. It allows a client program to build an optimization model, submit it to a Compute Server cluster (through a Cluster Manager), and later check on the status of the model and retrieve its solution. For more information, please refer to the Batch Optimization section.

Commonly used methods on batch objects include update (refresh attributes from the Cluster Manager), abort (abort execution of a batch request), retry (retry optimization for an interrupted or failed batch), discard (remove the batch request and all related information from the Cluster Manager), and getJSONSolution (query solution information for the batch request).

These methods are built on top of calls to the Cluster Manager REST API. They are meant to simplify such calls, but note that you always have the option of calling the REST API directly.

Batch objects have four attributes:

- BatchID: Unique ID for the batch request.
- BatchStatus: Last batch status.
- BatchErrorCode: Last error code.
- BatchErrorMessage: Last error message.

You can access their values as you would for other attributes: `batch.BatchStatus`, `batch.BatchID`, etc. Note that all Batch attributes are locally cached, and are only updated when you create a client-side batch object or when you explicitly update this cache, which can done by calling update.

Batch()

```
Batch ( batchID, env )
```

Given a `BatchID`, as returned by optimizeBatch, and a Gurobi environment that can connect to the appropriate Cluster Manager (i.e., one where parameters CSManager, UserName, and ServerPassword have been set appropriately), this function returns a Batch object. With it, you can query the current status of the associated batch request and, once the batch request has been processed, you can query its solution. Please refer to the Batch Optimization section for details and examples.

Arguments:

`batchID`: ID of the batch request for which you want to access status and other information.

`env`: The environment in which the new batch object should be created.

Return value:

New batch object.

Example usage:

```
batch = gp.Batch(batchID, env)

# Automatically disposed with context manager
with gp.Batch(batchID, env) as batch:
  pass
```

Batch.abort()

```
abort ( )
```

This method instructs the Cluster Manager to abort the processing of this batch request, changing its status to ABORTED. Please refer to the Batch Status Codes section for further details.

Example usage:

```
starttime = time.time()
while batch.BatchStatus == GRB.BATCH_SUBMITTED:
    # Abort this batch if it is taking too long
    curtime = time.time()
    if curtime - starttime > maxwaittime:
        batch.abort()
        break

    # Wait for two seconds
    time.sleep(2)

    # Update the resident attribute cache of the Batch object with the
    # latest values from the cluster manager.
    batch.update()

    # If the batch failed, we retry it
```

Batch.discard()

```
discard ( )
```

This method instructs the Cluster Manager to remove all information related to the batch request in question, including the stored solution if available. Further queries for the associated batch request will fail with error code GRB_ERROR_DATA_NOT_AVAILABLE. Use this function with care, as the removed information can not be recovered later on.

Example usage:

```
# Remove batch request from manager
batch.discard()
```

Batch.dispose()

```
dispose ( )
```

Free all resources associated with this Batch object. After this method is called, this Batch object must no longer be used.

Example usage:

```
batch.dispose()
```

Batch.getJSONSolution()

```
getJSONSolution ( )
```

This method retrieves the solution of a completed batch request from a Cluster Manager. The solution is returned as a JSON solution string. For this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Example usage:

```
print("JSON solution:")
# Get JSON solution as string, create dict from it
sol = json.loads(batch.getJSONSolution())
```

Batch.retry()

```
retry ( )
```

This method instructs the Cluster Manager to retry optimization of a failed or aborted batch request, changing its status to `SUBMITTED`. Please refer to the Batch Status Codes section for further details.

Example usage:

```
    break

# Wait for two seconds
time.sleep(2)

# Update the resident attribute cache of the Batch object with the
# latest values from the cluster manager.
batch.update()

# If the batch failed, we retry it
if batch.BatchStatus == GRB.BATCH_FAILED:
    batch.retry()
```

Batch.update()

```
update ( )
```

All Batch attribute values are cached locally, so queries return the value received during the last communication with the Cluster Manager. This method refreshes the values of all attributes with the values currently available in the Cluster Manager (which involves network communication).

Example usage:

```
# Update the resident attribute cache of the Batch object with the
# latest values from the cluster manager.
batch.update()
```

Batch.writeJSONSolution()

```
writeJSONSolution ( filename )
```

This method returns the stored solution of a completed batch request from a Cluster Manager. The solution is returned in a gzip-compressed JSON file. The file name you provide must end with a `.json.gz` extension. The JSON format is described in the JSON solution format section. Note that for this call to succeed, the status of the batch request must be `COMPLETED`. Please refer to the Batch Status Codes section for further details.

Arguments:

`filename`: Name of file where the solution should be stored (in JSON format).

Example usage:

```
batch.writeJSONSolution('batch-sol.json.gz')
```

6.20 GRB

Class for Python constants. Classes GRB.Attr and GRB.Param are used to get or set Gurobi attributes and parameters, respectively.

Constants

The following list contains a set of constants that are used by the Gurobi Python interface. You would refer to them using a `GRB.` prefix (e.g., `GRB.OPTIMAL`).

```
# Status codes

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

# Batch status codes

  BATCH_CREATED   = 1
  BATCH_SUBMITTED = 2
  BATCH_ABORTED   = 3
  BATCH_FAILED    = 4
  BATCH_COMPLETED = 5

# Version number

  VERSION_MAJOR     = 9
  VERSION_MINOR     = 0
  VERSION_TECHNICAL = 1

# Basis status

  BASIC          = 0
  NONBASIC_LOWER = -1
  NONBASIC_UPPER = -2
  SUPERBASIC     = -3

# Constraint senses

  LESS_EQUAL    = '<'
  GREATER_EQUAL = '>'
  EQUAL         = '='

# Variable types
```

```
  CONTINUOUS = 'C'
  BINARY     = 'B'
  INTEGER    = 'I'
  SEMICONT   = 'S'
  SEMIINT    = 'N'

# Objective sense

  MINIMIZE = 1
  MAXIMIZE = -1

# SOS types

  SOS_TYPE1 = 1
  SOS_TYPE2 = 2

# General constraint types

  GENCONSTR_MAX       = 0
  GENCONSTR_MIN       = 1
  GENCONSTR_ABS       = 2
  GENCONSTR_AND       = 3
  GENCONSTR_OR        = 4
  GENCONSTR_INDICATOR = 5
  GENCONSTR_PWL       = 6
  GENCONSTR_POLY      = 7
  GENCONSTR_EXP       = 8
  GENCONSTR_EXPA      = 9
  GENCONSTR_LOG       = 10
  GENCONSTR_LOGA      = 11
  GENCONSTR_POW       = 12
  GENCONSTR_SIN       = 13
  GENCONSTR_COS       = 14
  GENCONSTR_TAN       = 15

# Numeric constants

  INFINITY  = 1e100
  UNDEFINED = 1e101
  MAXINT    = 2000000000

# Limits

  MAX_NAMELEN    = 255
  MAX_STRLEN     = 512
  MAX_TAGLEN     = 10240
  MAX_CONCURRENT = 64

# Other constants

  DEFAULT_CS_PORT = 61000

# Errors

  ERROR_OUT_OF_MEMORY            = 10001
  ERROR_NULL_ARGUMENT              = 10002
  ERROR_INVALID_ARGUMENT           = 10003
  ERROR_UNKNOWN_ATTRIBUTE          = 10004
  ERROR_DATA_NOT_AVAILABLE         = 10005
  ERROR_INDEX_OUT_OF_RANGE         = 10006
  ERROR_UNKNOWN_PARAMETER          = 10007
  ERROR_VALUE_OUT_OF_RANGE         = 10008
  ERROR_NO_LICENSE                 = 10009
  ERROR_SIZE_LIMIT_EXCEEDED        = 10010
  ERROR_CALLBACK                   = 10011
  ERROR_FILE_READ                  = 10012
  ERROR_FILE_WRITE                 = 10013
  ERROR_NUMERIC                    = 10014
  ERROR_IIS_NOT_INFEASIBLE         = 10015
  ERROR_NOT_FOR_MIP                = 10016
  ERROR_OPTIMIZATION_IN_PROGRESS = 10017
  ERROR_DUPLICATES                 = 10018
  ERROR_NODEFILE                   = 10019
  ERROR_Q_NOT_PSD                  = 10020
  ERROR_QCP_EQUALITY_CONSTRAINT  = 10021
  ERROR_NETWORK                    = 10022
  ERROR_JOB_REJECTED               = 10023
  ERROR_NOT_SUPPORTED              = 10024
  ERROR_EXCEED_2B_NONZEROS         = 10025
  ERROR_INVALID_PIECEWISE_OBJ      = 10026
  ERROR_UPDATEMODE_CHANGE          = 10027
  ERROR_CLOUD                      = 10028
  ERROR_MODEL_MODIFICATION         = 10029
  ERROR_CSWORKER                   = 10030
  ERROR_TUNE_MODEL_TYPES           = 10031
  ERROR_SECURITY                   = 10032
  ERROR_NOT_IN_MODEL               = 20001
  ERROR_FAILED_TO_CREATE_MODEL   = 20002
  ERROR_INTERNAL                   = 20003
```

GRB.Attr

The constants defined in this class are used to get or set attributes (through Model.getAttr or Model.setAttr, for example). Please refer to the Attributes section to see a list of all attributes and their functions. You refer to an attribute using a `GRB.Attr` prefix (e.g., `GRB.Attr.Obj`). Note that these constants are simply strings, so wherever you might use this constant, you also have the option of using the string directly (e.g, `'obj'` rather than `GRB.Attr.Obj`).

GRB.Param

The constants defined in this class are used to get or set parameters Model.getParamInfo or Model.setParam. Please refer to the Parameters section to see a list of all parameters and their functions. You refer to a parameter using a `GRB.Param` prefix (e.g., `GRB.Param.displayInterval`). Note that these constants are simply strings, so wherever you might use this constant, you also have the option of using the string directly (e.g, `'displayInterval'` rather than `GRB.Param.-displayInterval`).

6.21 tuplelist

Gurobi tuple list. This is a sub-class of the Python `list` class that is designed to efficiently support a usage pattern that is quite common when building optimization models. In particular, if a `tuplelist` is populated with a list of tuples, the select function on this class efficiently selects tuples whose values match specified values in specified tuple fields. To give an example, the statement `l.select(1, '*', 5)` would select all member tuples whose first field is equal to '1' and whose third field is equal to '5'. The `'*'` character is used as a wildcard to indicate that any value is acceptable in that field.

You generally build `tuplelist` objects in the same way you would build standard Python lists. For example, you can use the `+=` operator to append a new list of items to an existing `tuplelist`, or the `+` operator to concatenate a pair of `tuplelist` objects. You can also call the `append`, `extend`, `insert`, `pop`, and `remove` functions.

To access the members of a `tuplelist`, you also use standard list functions. For example, `l[0]` returns the first member of a `tuplelist`, while `l[0:10]` returns a `tuplelist` containing the first ten members. You can also use `len(l)` to query the length of a list.

Note that `tuplelist` objects build and maintain a set of internal data structures to support efficient `select` operations. If you wish to reclaim the storage associated with these data structures, you can call the clean function.

A `tuplelist` is designed to store tuples containing scalar values (`int`, `float`, `string`, ...). It may produce unpredictable results with other Python objects, such as tuples of tuples. Thus, you can store `(1, 2.0, 'abc')` in a `tuplelist`, but you shouldn't store `((1, 2.0), 'abc')`.

tuplelist()

```
tuplelist ( list )
```

tuplelist constructor.

Arguments:

`list`: Initial list of member tuples.

Return value:

A tuplelist object.

Example usage:

```
l = tuplelist([(1,2), (1,3), (2,4)])
l = tuplelist([('A', 'B', 'C'), ('A', 'C', 'D')])
```

tuplelist.select()

```
select ( pattern )
```

Returns a `tuplelist` containing all member tuples that match the specified pattern. The pattern requires one argument for each field in the member tuple. A scalar argument must match the corresponding field exactly. A list argument matches if any list member matches the corresponding field. A `'*'` argument matches any value in the corresponding field.

Arguments:

`pattern`: Pattern to match for a member tuple.

Example usage:

```
l.select(1, 3, '*', 6)
l.select([1, 2], 3, '*', 6)
l.select('A', '*', 'C')
```

tuplelist.clean()

```
clean ( )
```

Discards internal data structure associated with a `tuplelist` object. Note that calling this routine won't affect the contents of the `tuplelist`. It only affects the memory used and the performance of later calls to select.

Example usage:

```
l.clean()
```

tuplelist.__contains__()

```
__contains__ ( val )
```

Provides efficient support for the Python `in` operator.

Example usage:

```
if (1,2) in l:
  print("Tuple (1,2) is in tuplelist l")
```

6.22 tupledict

Gurobi tuple dict. This is a sub-class of the Python `dict` class that is designed to efficiently support a usage pattern that is quite common when building optimization models. In particular, a `tupledict` is a Python `dict` where the keys are stored as a Gurobi tuplelist, and where the values are typically Gurobi Var objects. Objects of this class make it easier to build linear expressions on sets of Gurobi variables, using tuplelist.select() syntax and semantics.

You typically build a `tupledict` by calling Model.addVars. Once you've created a `tupledict` `d`, you can use `d.sum()` to create a linear expression that captures the sum of the variables in the `tupledict`. You can also use a command like `d.sum(1, '*', 5)` to create a sum over a subset of the variables in `d`. Assuming the keys for the `tupledict` are tuples containing three fields, this statement would create a linear expression that captures the sum over all variables in `d` whose keys contain a 1 in the first field of the tuple and a 5 in the third field (the `'*'` character is a wildcard that indicates that any value is acceptable in that field). You can also use `d.prod(coeff)` to create a linear expression where the coefficients are pulled from the dictionary `coeff`. For example, if `d(1,2,5)` contains variable `x` and `coeff(1,2,5)` is 2.0, then the resulting expression would include term $2.0 * x$.

To access the members of a `tupledict`, you can use standard dict indexing. For example, `d[1,2]` returns the value associated with tuple `(1,2)`.

Note that a `tupledict` key must be a tuple of scalar values (`int`, `float`, `string`, ...). Thus, you can use `(1, 2.0, 'abc')` as a key, but you can't use `((1, 2.0), 'abc')`.

Note that `tupledict` objects build and maintain a set of internal data structures to support efficient `select` operations. If you wish to reclaim the storage associated with these data structures, you can call the clean function.

tupledict()

```
tupledict  ( args, kwargs )
```

`tupledict` constructor. Arguments are identical to those of a Python `dict` constructor. Note that you will typically use Model.addVars to build a `tupledict`.

Arguments:

`args`: Positional arguments.

`kwargs`: Named arguments.

Return value:

A `tupledict` object.

Example usage:

```
d = tupledict([((1,2), 'onetwo'), ((1,3), 'onethree'), ((2,3), 'twothree')])
print(d[1,2]) # prints 'onetwo'
```

tupledict.select()

```
select  ( pattern )
```

Returns a `list` containing the values associated with keys that match the specified tuple pattern. The pattern should provide one value for each field in the key tuple. A `'*'` value indicates that any value is accepted in that field.

Without arguments, this method returns a list of all values in the `tupledict`.

Arguments:

`pattern`: Pattern to match for a key tuple.

Example usage:

```
d = tupledict([((1,2), 'onetwo'), ((1,3), 'onethree'), ((2,3), 'twothree')])
print(d.select())       # prints ['onetwo', 'onethree', 'twothree']
print(d.select(1, '*')) # prints ['onetwo', 'onethree']
print(d.select('*', 3)) # prints ['onethree', 'twothree']
print(d.select(1, 3))   # prints ['onethree']
```

tupledict.sum()

```
sum ( pattern )
```

Returns the sum of the values associated with keys that match the specified pattern. If the values are Gurobi `Var` objects, the result is a `LinExpr`. The pattern should provide one value for each field in the key tuple. A `'*'` value indicates that any value is accepted in that field.

Without arguments, this method returns the sum of all values in the `tupledict`.

Arguments:

`pattern`: Pattern to match for a key tuple.

Example usage:

```
x = m.addVars([(1,2), (1,3), (2,3)])
expr = x.sum()       # LinExpr: x[1,2] + x[1,3] + x[2,3]
expr = x.sum(1, '*') # LinExpr: x[1,2] + x[1,3]
expr = x.sum('*', 3) # LinExpr: x[1,3] + x[2,3]
expr = x.sum(1, 3)   # LinExpr: x[1,3]
```

tupledict.prod()

```
prod ( coeff, pattern )
```

Returns a linear expression that contains one term for each tuple that is present in both the `tupledict` and in the argument `dict`. For example, `x.prod(coeff)` would contain term $2.0 * var$ if `x[1,2] = var` and `coeff[1,2] = 2.0`.

Arguments:

`coeff`: Python `dict` that maps tuples to coefficients.

`pattern`: Pattern to match for a key tuple.

Example usage:

```
x = m.addVars([(1,2), (1,3), (2,3)])
coeff = dict([((1,2), 2.0), ((1,3), 2.1), ((2,3), 3.3)])
expr = x.prod(coeff) # LinExpr: 2.0 x[1,2] + 2.1 x[1,3] + 3.3 x[2,3]
expr = x.prod(coeff, '*', 3) # LinExpr: 2.1 x[1,3] + 3.3 x[2,3]
```

tupledict.clean()

`clean ( )`

Discards internal data structure associated with a `tupledict` object. Note that calling this routine won't affect the contents of the `tupledict`. It only affects the memory used and the performance of later calls to select.

Example usage:

```
d.clean()
```

6.23 General Constraint Helper Functions

Gurobi general constraint helper functions - used in conjunction with overloaded operators and Model.addConstr or Model.addConstrs to build general constraints.

abs_()

```
abs_ ( variable )
```

Used to set a decision variable equal to the absolute value of another decision variable.

Example usage:

```
m.addConstr(y == abs_(x))
```

Return value:

Returns a GenExpr object.

and_()

```
and_ ( variables )
```

Used to set a binary decision variable equal to the logical AND of a list of other binary decision variables. You can pass the arguments as a Python list or as a comma-separated list.

Note that the Gurobi Python interface includes an equivalent `all_()` function.

Example usage:

```
m.addConstr(z == and_(x, y))
m.addConstr(z == and_([x, y]))
```

Return value:

Returns a GenExpr object.

max_()

```
max_ ( variables )
```

Used to set a decision variable equal to the maximum of a list of decision variables (or constants). You can pass the arguments as a Python list or as a comma-separated list.

Example usage:

```
m.addConstr(z == max_(x, y, 3))
m.addConstr(z == max_([x, y, 3]))
```

Return value:

Returns a GenExpr object.

min_()

```
min_ ( variables )
```

Used to set a decision variable equal to the minimum of a list of decision variables (or constants). You can pass the arguments as a Python list or as a comma-separated list.

Example usage:

```
m.addConstr(z == min_(x, y, 3))
m.addConstr(z == min_([x, y, 3]))
```

Return value:

Returns a GenExpr object.

or_()

```
or_ ( variables )
```

Used to set a binary decision variable equal to the logical OR of a list of other binary decision variables. You can pass the arguments as a Python list or as a comma-separated list.

Note that the Gurobi Python interface includes an equivalent `any_()` function.

Example usage:

```
m.addConstr(z == or_(x, y))
m.addConstr(z == or_([x, y]))
```

Return value:

Returns a GenExpr object.