Gurobi Guidelines for Numerical Issues

February 2017

Background

Models with numerical issues can lead to undesirable results: slow performance, wrong answers or inconsistent behavior. When solving a model with numerical issues, tiny changes in the model or computer can make a big difference in the results. For example, changing one coefficient can make Gurobi Optimizer much slower, or upgrading to a new computer could cause Gurobi Optimizer to report an incorrect solution. This behavior is not a defect in Gurobi Optimizer: any solver will face the same issue.

This document is intended as a practical guide to managing models with numerical issues. A rigorous mathematical discussion is beyond the scope of this guide. However, we will cover some basic concepts in this section.

Computers use finite-precision arithmetic, so a simple arithmetic calculation like 1/3 produces a value with a fixed number of digits, such as `0.3333333333333333`. This decimal value is not exactly 1/3, so we get a small amount of round-off error. Gurobi Optimizer is designed to be robust, so a small error like this does not affect the result. However, you can get more significant round-off error if you approximate 1/3 by a value like 0.3333. You can also get unexpected results with values that are very large or very close to zero. For example, the following constraints:

$$\begin{array}{lcc} x & \leq & 1000000y \\ x & \geq & 0 \\ y & & \text{binary} \end{array}$$

are a common method to represent a fixed charge when $x > 0$. Here, the values $x = 9.9999$, $y = 0.0000099999$ are feasible with respect to the default integer feasibility tolerance (`IntFeasTol=1e-5`). This would allow x to be positive without incurring an expensive fixed charge on y. Furthermore, you cannot completely avoid this issue by adjusting the `IntFeasTol` parameter.

Consider the simplex method for linear programming, a core algorithm in Gurobi Optimizer. When solving a linear program, the constraints can be written in matrix form as:

$$Ax = b$$

This can be partitioned into basic and nonbasic variables:

$$Bx_B + Nx_N = b$$

and the simplex method solves for the basic variables x_B:

$$x_B = B^{-1}(b - Nx_N)$$

Gurobi Optimizer does not explicitly invert the B matrix; it solves for x_B via matrix multiplication and factorization. If the B matrix is *ill-conditioned*, small changes in data can make big changes in solution values. In each iteration, Gurobi Optimizer solves for x_B. With many iterations, errors can accumulate to produce slow or incorrect solutions to an optimization model.

If these matrix concepts are new to you, consider a very simple case: a 1x1 matrix $B = [m]$, where m is simply a number. The inverse $B^{-1} = [1/m]$. In this simple example, a value of m close to zero would make a large value of $1/m$; this matrix B would be ill-conditioned. The problem is that a small change or error in input data will become a very large change in the result when you multiply by this large value $1/m$.

Furthermore, other optimization algorithms are more numerically sensitive than LP simplex. Thus, you are more likely to get numerical issues when solving a large model using barrier, or when solving a quadratic program; you must be extra careful when formulating and solving these models.

In the rest of this guide, we will learn:

- To identify if a model contains numerical issues
- To reduce numerical issues by reformulating a model
- To manage numerical issues via Gurobi parameters

Does a model have numerical issues?

Not all performance issues are caused by numerical issues. Here are some steps to determine whether a model has numerical issues:

1. Isolate the model for testing by exporting a model file and a parameter file. The easiest way to do this is to create a gurobi.env file in your working directory that contains the following line:

   ```
   GURO_PAR_DUMP 1
   ```

   Then, run your Gurobi program, which will produce gurobi.rew and gurobi.prm files. Afterwards, you should delete the gurobi.env file so that other tests do not overwrite these files.

2. Using the Gurobi Interactive shell, run some simple Python code to read the model file and print the summary statistics:

   ```
   m = read("gurobi.rew")
   m.printStats()
   ```

   The output will look like:

   ```
   Linear constraint matrix    : 67069 Constrs, 128001 Vars, 350728 NZs
   Matrix coefficient range    : [ 0.017648, 11.2588 ]
   Objective coefficient range : [ 0.00195312, 0.00195312 ]
   Variable bound range        : [ 200, 1000 ]
   RHS coefficient range       : [ 1.58, 1000 ]
   ```

The range of numerical coefficients is one indication of potential numerical issues. As a very rough guideline, the ratio of the largest to the smallest coefficient should be less than 1e9 (10^9); smaller is better. In this example, the matrix range is 11.2588 / 0.017648 = 637.9646418858.

3. If possible, solve the model using your parameters and review the logs. With the Python shell, use code like the following:

```
m.read("gurobi.prm")
m.optimize()
```

Here are some examples of warning messages that suggest numerical issues:

```
Warning: Markowitz tolerance tightened to 0.5
Warning: switch to quad precision
Numeric error
Numerical trouble encountered
Restart crossover...
Sub-optimal termination
Warning: ... variables dropped from basis
Warning: unscaled primal violation = ... and residual = ...
Warning: unscaled dual violation = ... and residual = ...
```

4. If you finish running the optimize function, print the solution statistics. With the Python shell, use code like the following:

```
m.printQuality()
```

Large violations are another indicator of numerical issues. Also, for a pure LP (without integer variables), print the condition number via the following Python command:

```
m.Kappa
```

The condition number measures the potential for error in linear calculations; a large condition number like 1e8 (10^8) is another indication of numerical issues.

If these tests identify that the model has numerical issues, the remainder of this document should be helpful. If not, try traditional methods of improving Gurobi performance, such as the grbtune parameter tuning tool and the Parameter Guidelines section of the Gurobi Optimizer Reference Manual.

Reformulating a model to reduce numerical issues

The best way to handle numerical issues is to reformulate the model. The phrase "garbage in, garbage out" applies to optimization: it is far better to fix numerical issues in the model than to make the solver adjust for them. Here are some general guidelines:

1. Try to have a reasonable range of model coefficients. For example, a model containing some coefficients of 10^{-5} and others of 10^6 has a range of $10^6 / 10^{-5} = 10^{11}$, which can be numerically challenging. As a rough guideline, the range should be smaller than 10^8.

2. Avoid overly large big-M coefficients in logical expressions. Consider:

$$\begin{array}{lll} x & \leq & My \\ y & & \text{binary} \end{array}$$

If you know that x is always less than 1000, then set $M = 1000$. Alternately, you can avoid the big-M value by using a *general constraint*, a new feature added in Gurobi Optimizer 7.0. This example is equivalent to the indicator constraint $y = 0 \rightarrow x = 0$.

3. Avoid overly large penalty terms. If a penalty term represents an undesirable decision, it's best to use an accurate estimate of the costs of that undesirable alternative. In other cases, large penalty terms represent hierarchies in the objective function. Gurobi Optimizer 7.0 added a feature for multiple objectives. There are two approaches: blended objectives and hierarchical objectives. Use the hierarchical objective feature to express each goal as a distinct objective function, without penalty terms. For details, see the Multiple Objectives section of the Gurobi Optimizer Reference Manual.

4. Compute coefficients precisely. For example, 0.3333 is a much worse representation of 1/3 than `0.3333333333333333`. Alternately, eliminate round-off by multiplying coefficients in a constraint by the least common multiple. When coefficients are not precise, you introduce unnecessary error in calculations, which can contribute to numerical issues. Furthermore, the staff at Gurobi do not recommend relaxing numerical tolerances as a workaround for round-off errors. Always use double-precision numerical coefficients, and compute all coefficients precisely.

Solver parameters to manage numerical issues

Reformulating a model may not always be possible, or it may not completely resolve numerical issues. Thus, when you must solve a model that has numerical issues, some Gurobi parameters can be helpful. Let's discuss them in order of what you should try.

Presolve

Gurobi presolve algorithms are designed to make a model smaller and easier to solve. However, in some cases, presolve can contribute to numerical issues. The following Python code can help you determine if this is happening. First, read the model file and print the summary statistics for the presolved model:

```
m = read("gurobi.rew")
p = m.presolve()
p.printStats()
```

If the numerical range looks much worse than the original model, try the parameter Aggregate=0:

```
m.reset()
m.Params.Aggregate = 0
p = m.presolve()
p.printStats()
```

If that is also numerically problematic, you may need to disable presolve completely using the parameter `Presolve=0`; try the steps above using `m.Params.Presolve = 0`.

If the statistics look better with Aggregate=0 or Presolve=0, you should test these parameters. For a continuous (LP) model, you can test them directly. For a MIP, you should compare the LP relaxation with and without these parameters. The following Python commands create three LP relaxations: the model without presolve, the model with presolve, and the model with Aggregate=0:

```
m = read("gurobi.rew")
r = m.relax()
r.write("gurobi.relax-nopre.rew")
p = m.presolve()
r = p.relax()
r.write("gurobi.relax-pre.rew")
m.reset()
m.Params.Aggregate = 0
p = m.presolve()
r = p.relax()
r.write("gurobi.relax-agg0.rew")
```

With these three files, use the techniques mentioned earlier to determine if Presolve=0 or Aggregate=0 improves the numerics of the LP relaxation.

Finally, if Aggregate=0 helps numerics but makes the model too slow, try AggFill=0 instead.

Choosing the right algorithm

Gurobi Optimizer provides two main algorithms to solve continuous models and the continuous relaxation of mixed-integer models. The barrier algorithm is usually fastest for large, difficult models. However, the barrier algorithm is also the most numerically sensitive algorithm. Even when the barrier algorithm converges, numerical issues can cause crossover to stall.

The simplex method is a good alternative since it is generally less sensitive to numerical issues. To use dual simplex or primal simplex, set the parameter Method=1 or Method=0, respectively.

However, in many optimization applications, not all data sets have numerical issues. If you use simplex exclusively, the application may be slow with data that do not have numerical issues. In this case, you should use the concurrent optimizer, which uses multiple algorithms simultaneously and returns the solution from the first one to finish. The easy way to use the concurrent optimizer is to set the parameter Method=3 or Method=4. For full control over the concurrent optimizer, you can create concurrent environments, where you can set specific algorithmic parameters for each concurrent solve. For example, you can create one concurrent environment with Method=0 and another with Method=1 to use primal and dual simplex simultaneously. Finally, you can use concurrent optimization with either a single computer or in distributed optimization with multiple computers. With a single computer, the different algorithms run on multiple threads, each using processor cores in the computer. With distributed optimization, independent computers run the separate algorithms, which can be faster since the computers do not share memory.

Making the algorithm less sensitive

When all else fails, try the following parameters to make the algorithms more robust:

Parameter	Applies to	Comments
`ScaleFlag`, `ObjScale`	All models	It is *always* best to reformulate a model yourself. However, in cases when that is not possible, these two parameters provide some of the same benefits. Set `ScaleFlag=2` for aggressive scaling of the coefficient matrix. `ObjScale` rescales the objective row; a negative value will automatically rescale the objective in terms of its largest coefficient. For example, `ObjScale=-0.5` will divide all objective coefficients by the square root of the largest objective coefficient.
`NumericFocus`	All models	The `NumericFocus` parameter controls how the solver manages numerical issues. By default, the solver focuses on speed. Settings 1-3 increasingly shift the focus towards more care in numerical computations. With higher values, it will spend more time checking the numerical accuracy of intermediate results, which can make it slower. `NumericFocus` automatically controls quad precision and the Markowitz tolerance, so it is generally sufficient to try the different values of `NumericFocus`. However, when `NumericFocus` helps numerics but makes everything much slower, try setting `Quad=1` and/or larger values of `MarkowitzTol` such as `0.1` or `0.5`.
`NormAdjust`	Simplex	In some cases, the solver can be more robust with different values of the simplex pricing norm. Try setting `NormAdjust` to `0`, `1`, `2` or `3`.
`BarHomogeneous`	Barrier	For models that are infeasible or unbounded, the default barrier algorithm may have numerical issues. Try setting `BarHomogeneous=1`.
`CrossoverBasis`	Barrier	Setting `CrossoverBasis=1` takes more time but can be more robust when creating the initial crossover basis.
`GomoryPasses`	MIP	In some MIP models, Gomory cuts contribute to numerical issues. Setting `GomoryPasses=0` may help numerics, but it can make the MIP more difficult to solve.
`Cuts`	MIP	In some MIP models, various cuts contribute to numerical issues. Setting `Cuts=1` or `Cuts=0` may help numerics, but it may make the MIP more difficult to solve.

Tolerance values (`FeasibilityTol`, `OptimalityTol`, `IntFeasTol`) are generally *not* helpful for numerical issues. Numerical issues can be better handled by reformulating a model.

Further reading

"Floating point". Wikipedia, https://en.wikipedia.org/wiki/Floating_point.

Michael L. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM, 2001.