Gurobi Guidelines for Numerical Issues

Numerical instability is a generic label often applied to situations where solving an optimization model produces results that are erratic, inconsistent, or unexpected, or when the underlying algorithms exhibit poor performance or are unable to converge. There are many potential causes of this behavior; however, most can be grouped into four categories:

- Rounding coefficients while building the model.
- Limitations of floating-point arithmetic.
- Unrealistic expectations about achievable precision.
- Ill conditioning, or geometry-induced issues.

This section explains these issues and how they affect both performance and solution quality. We also provide some general rules and some advanced techniques to help avoid them. Although we will treat each of these four sources separately, it is important to remember that their effects often feed off of each other. We also provide tips on how to diagnose numerical instability in your models.

Finally, we discuss the Gurobi parameters that can be modified to improve solution accuracy. We should stress now, however, that the best way to improve numerical behavior and performance is to reformulate your model. Parameters can help to manage the effects of numerical issues, but there are limits to what they can do, and they typically come with a substantial performance cost.

24.1 Avoid rounding of input

A common source of numerical issues is numerical rounding in the numbers that are used to represent constraint matrix coefficients. To illustrate the issue, consider the following example:

$$\begin{aligned} x - 6y &= 1 \\ 0.333x - 2y &= .333 \end{aligned}$$

It may be tempting to say that the two equations are equivalent, but adding both to a model will lead to an incorrect result. This is an important point for our users: Gurobi will always trust the input numbers that they provide, and will never change them unless the change can be shown to not affect the solution.

So, with this in mind, during presolve Gurobi can use the second constraint to determine:

$$y := 0.1665x - 0.1665$$

When substituted into the first constraint, this yields

$$\begin{aligned} x - 6 \cdot (0.1665x - 0.1665) &= 1 \\ \Leftrightarrow 0.001x &= 0.001 \end{aligned}$$

and thus $x = 1,\ y = 0$ as the only solution.

If user had provided these two equations instead:

$$\begin{aligned} x - 6y &= 1 \\ 0.3333333333333333x - 2y &= 0.3333333333333333 \end{aligned}$$

this would give:

$$y := 0.1666666666666667x - 0.1666666666666667$$

which yields:

$$\begin{aligned} x - 6 \cdot (0.1666666666666667x - 0.1666666666666667) &= 1 \\ \Leftrightarrow 2 \cdot 10^{-16}x + 1 + 2 \cdot 10^{-16} &\approx 1 \end{aligned}$$

Even with a very small threshold for treating a coefficient as zero, the result here is that the first constraint is truly redundant. Any solution with $x = 6y + 1$ would be accepted as feasible.

The main point is that constraints that are exactly parallel, or linearly dependant (within double-precision floating-point and small tolerances) are harmless, but constraints that are almost parallel to each other produce tiny coefficients in the linear system solves and in preprocessing, which can wreak havoc on the solution process. In the next section, we expand on the limits *double-precision floating-point* numbers, and in particular why $1 \approx 1 + 2 \cdot 10^{-16}$.

24.2 Real numbers are not real

To say that real numbers aren't real is not just a play on words, but a computational reality. Let's do a simple experiment: try the following in your favorite number-crunching tool. In Excel:

```
=IF(1+1E-016 = 1,1,0)
```

will print 1. In Python:

```
>>> 1 == 1+1e-16
True
```

In C, the code

```
#include<stdio.h>
int main(void)
{
  if (1+1e-16 == 1) printf("True\n");
  else              printf("False\n");
  return 0;
}
```

will print `True`. In R:

```
> 1 == 1+1e-16
[1] TRUE
```

Note that this behavior is not restricted to *small* numbers; it also happens with larger numbers. For example:

```
>>> 1+1e16 == 1e16
True
```

This shows that the *precision* of the result depends on the relative scale of the involved numbers.

Although this behavior is typical, there are some exceptions. One is the GNU-bc command line tool:

```
> bc
1.0 == 1.0+10^(-16)
1
scale=20
1.0 == 1.0+10^(-16)
0
1.0 == 1.0+10^(-21)
1
```

When we set the `scale` parameter to 20, the code is able to recognize that the numbers are different. This just shifts the bar, though; `bc` still fails to recognize the difference between the last two numbers. Another library that allows for extended, or even *unlimited* (up to memory) precision is the GNU Multiple Precision Arithmetic Library, but its details are beyond the scope of this document.

The reason for these *failures* is that computers must store numbers as a sequence of bits, and most common implementations adhere to the IEEE 754 standard. In particular, IEEE-754 sets the standard for *double-precision* format. This standard is so pervasive that almost all computers have specialized hardware to improve performance for operations on numbers represented as such. One consequence is that mathematical operations on alternative extended number representations tend to be significantly slower than operations on numbers represented following the IEEE 754 standard. Degradation of 10X or even 100X are common.

Due to the performance obtained from hardware support for double-precision arithmetic, Gurobi relies on this standard (as does most software). However, this speed comes at a cost: computed results often differ from what mathematics may dictate. For example, the associative property (`a + (b + c) = (a + b) + c`) is a fundamental property of arithmetic, but double-precision arithmetic gives (in Python):

```
>>> (1+1e-16)+1e-16 == 1 + (1e-16 + 1e-16)
False
```

Furthermore, many common numbers (e.g. 0.1) cannot be represented exactly.

Consequently, simple questions like whether two numbers are equal, or whether a number is equal zero, or whether a number is integral, can be quite complicated when using floating-point arithmetic.

24.3 Tolerances and user-scaling

Gurobi will solve the model as defined by the user. However, when evaluating a candidate solution for feasibility, in order to account for possible round-off errors in the floating-point evaluations, we must allow for some *tolerances*.

To be more precise, satisfying *Optimality Conditions* requires us to test at least the following three criteria:

`IntFeasTol`: Integrality of solutions, i.e., whether a integer variable x takes an integer value or not. More precisely, x will be considered *integral* if `abs(x - floor(x + 0.5))` $\leq$ `IntFeasTol`.

`FeasibilityTol`: Feasibility of primal constraints, i.e., whether $a \cdot x \leq b$ holds for the *primal* solution. More precisely, $a \cdot x \leq b$ will be considered to hold if `(a * x) - b` $\leq$ `FeasibilityTol`.

`OptimalityTol`: Feasibility of dual constraints, i.e., whether $a \cdot y \leq c$ holds for the *dual* solution. More precisely, $a \cdot y \leq c$ will be considered to hold if `(a * y) - c` $\leq$ `OptimalityTol`.

Note that these tolerances are **absolute**; they do not depend on the scale of the quantities involved in the computation. This means that when formulating a problem, these tolerances should be taken into account, specially to select the units in which variables and constraints will be expressed.

It is very important to note that the usage of these *tolerances* implicitly defines a *gray zone* in the search space in which solutions that are very slightly infeasible can still be accepted as feasible. However, the solver will not explicitly search for such solutions.

For this reason, it is actually possible (although highly unlikely for well-posed problems) for a model to be reported as being both *feasible* and *infeasible* (in the sense stated above). This can occur if the model is infeasible in exact arithmetic, but there exists a solution that is feasible within the solver tolerances. For instance, consider:

$$\begin{array}{lll} \min & 0 & \\ s.t. & x \leq & 0 \\ & x \geq & 10^{-10} \end{array}$$

Gurobi tolerances and the limitations of double-precision arithmetic

The default values for these primal and dual feasibility tolerances are 10^{-6}, and the default for the integrality tolerance is 10^{-5}. If you choose the range for your inequalities and variables correctly, you can typically ignore tolerance issues entirely.

To give an example, if your constraint right-hand side is on the order of 10^3, then relative numeric errors from computations involving the constraint (if any) are likely to be less than 10^{-9}, i.e., less than one in a billion. This is usually far more accurate than the accuracy of input data, or even of what can be measured in practice.

However, if you define a variable $x \in [-10^{-6}, 10^{-6}]$, then relative numeric error may be as big as 50% of the variable range.

If, on the other hand, you have a variable $x \in [-10^{10}, 10^{10}]$, and you are using default primal feasibility tolerances; then what you are really asking is for the relative numeric error (if any) to be less than 10^{-16}. However, this is beyond the limits of comparison for double-precision numbers. This implies that you are not allowing any round-off error at all when testing feasible solutions for this particular variable. And although this might sound as a good idea, in fact, it is really bad, as any round-off computation may result in your truly optimal solution being rejected as infeasible.

Why scaling and geometry is relevant

This section provides a simple example of how scaling problems can slow down problem solving and, in extreme cases, result in unexpected answers. Consider the problem:

$$(P) \max\{cx : Ax = b, l \leq x \leq u\}$$

and let D be a diagonal matrix where $D_{ii} > 0$, $\forall i$. In theory, solving (P) should be equivalent to solving the related problem (P_D):

$$(P_D) \max\{cDx' : ADx' = b, D^{-1}l \leq x' \leq D^{-1}u\}$$

However, in practice, the two models behave very differently. To demonstrate this, we use a simple script rescale.py that randomly rescales the columns of the model. Let's consider the impact of rescaling on the problem pilotnov.mps.bz2. Solving the original problem gives the following output:

```
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Coefficient statistics:
  Matrix range     [3e-06, 9e+06]
  Objective range  [3e-03, 1e+00]
  Bounds range     [6e-06, 7e+04]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 254 rows and 513 columns
Presolve time: 0.01s
Presolved: 721 rows, 1659 columns, 11454 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -3.2008682e+05   1.435603e+05   0.000000e+00      0s
    1137   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1137 iterations and 0.13 seconds
Optimal objective -4.497276188e+03
Kappa: 1.949838e+06
```

Note the log message regarding the matrix coefficient range in the log (which in this case shows a range of [3e-06, 9e+06]).

If we run rescale.py -f pilotnov.mps.bz2 -s 1e3 (randomly rescaling columns up or down by as much as 10^3), we obtain:

```
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Coefficient statistics:
  Matrix range     [5e-09, 1e+10]
  Objective range  [2e-06, 1e+03]
  Bounds range     [5e-09, 6e+07]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 100 rows and 255 columns
Presolve time: 0.00s
Presolved: 875 rows, 1917 columns, 11899 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -6.2117921e+32   7.026405e+31   6.211792e+02      0s
Extra 2 simplex iterations after uncrush
    1166   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1166 iterations and 0.15 seconds
Optimal objective -4.497276188e+03
Kappa: 2.341493e+18
```

This time, the optimization process takes a more iterations, and also, we get an extra warning:

```
Extra 2 simplex iterations after uncrush,
```

This indicates that extra simplex iterations were performed on the unpresolved model. Also, note the very large value for Kappa; its meaning will be discussed in this section.

If we run `rescale.py -f pilotnov.mps.bz2 -s 1e6`, we obtain:

```
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Coefficient statistics:
  Matrix range     [5e-12, 1e+13]
  Objective range  [2e-09, 1e+06]
  Bounds range     [5e-12, 5e+10]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
Warning: Model contains large bounds
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 103 rows and 252 columns
Presolve time: 0.01s
Presolved: 872 rows, 1920 columns, 11900 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -6.4093202e+34   7.254491e+31   6.409320e+04      0s
Extra 151 simplex iterations after uncrush
    1903   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1903 iterations and 0.23 seconds
Optimal objective -4.497276188e+03
Warning: unscaled primal violation = 0.171778 and residual = 0.00142752
Kappa: 5.729068e+12
```

Now we get a much larger number of extra simplex iterations, and more troublingly, we get a warning about the quality of the resulting solution:

```
Warning:  unscaled primal violation = 0.171778 and residual = 0.00142752,
```

This message indicates that the solver had trouble finding a solution that satisfies the default tolerances.

Finally, if we run `rescale.py -f pilotnov.mps.bz2 -s 1e8`, we obtain:

```
Optimize a model with 975 rows, 2172 columns and 13054 nonzeros
Coefficient statistics:
  Matrix range     [3e-13, 7e+14]
  Objective range  [2e-11, 1e+08]
  Bounds range     [5e-14, 1e+13]
  RHS range        [1e-05, 4e+04]
Warning: Model contains large matrix coefficient range
Warning: Model contains large bounds
         Consider reformulating model or setting NumericFocus parameter
         to avoid numerical issues.
Presolve removed 79 rows and 242 columns
Presolve time: 0.00s

Solved in 0 iterations and 0.00 seconds
Infeasible model
```

In this case, the optimization run terminates almost instantly, but with the unexpected `Infeasible` result.

As you can see, as we performed larger and larger rescalings, we continued to obtain the same optimal value, but there were clear signs that the solver struggled. We see warning messages, as well increasing iteration counts, runtimes, and `Kappa` values. However, once we pass a certain rescaling value, the solver is no longer able to solve the model and instead reports that it is `Infeasible`.

Note that this is not a bug in Gurobi. It has to do with changing the meaning of numbers depending on their range, the use of fixed tolerances, and in the changing geometry of the problem due to scaling. We will discuss this topic further in a later section.

Recommended ranges for variables and constraints

Keeping the lessons of the previous section in mind, we recommended that right-hand sides of inequalities representing physical quantities (even budgets) should be scaled so that they are on the order of 10^4 or less. The same applies to variable domains, as variable bounds are again linear constraints.

In the case of objective functions, we recommend that good solutions should have an optimal value that is less than 10^4, and ideally also above one (unless the objective coefficients are all zero). This is because the `OptimalityTol` is used to ensure that *reduced cost* are *close enough* to zero. If coefficients are too large, we again face difficulties in determining whether an LP solution truly satisfies the optimality conditions or not. On the other hand, if the coefficients are too small, then it may be too easy to satisfy the feasibility conditions.

The coefficients of the constraint matrix are actually more important than the right-hand side values, variable bounds, and objective coefficients mentioned here. We'll discuss those shortly.

Improving ranges for variables and constraints

There are three common ways to improve ranges for objectives, constraints and variables:

- Use problem-specific information to tighten bounds:

  Although presolve, and, in particular, bound strengthening, is quite good at deriving implied variables bounds, it may not have access to all of the information known to the modeler. Incorporating tighter bounds directly into the model can not only improve the numerical behavior, but it can also speed up the optimization process.

- Choose the right units to express your variables and constraints:

  When defining your variables and constraints, it is important to choose *units* that are consistent with tolerances. To give an example, a constraint with a 10^{10} right-hand side value is not going to work well with the default 10^{-6} feasibility tolerance. By changing the units (e.g., replacing pounds with tons, or dollars with millions of dollars, or ...), it is often possible to significantly improve the numerics of the problems.

- Disaggregate multiple objectives:

  A common source for very large range of objective coefficients is the practice of modeling hierarchical objectives as an aggregation of objective functions with large multipliers. For example, if the user wants to optimize a problem P with objective function $f_1(x)$ and then,

subject to $f_1(x)$ being optimal, optimize $f_2(x)$, a common trick is to use as surrogate objective $\bar{f}(x) = Mf_1(x) + f_2(x)$ where M is a large constant. When you combine a large M with a relatively tight dual feasibility tolerance, it becomes much harder for the solver to find solutions that achieve dual feasibility. We recommend that you either use as small a constant M as possible or reformulate your model using a hierarchical objective (which is made easier by our multi-objective optimization features).

These techniques are usually sufficient to eliminate the problems that arise from bad scaling.

Advanced user scaling

In the previous sections, we presented some simple strategies to limit the ranges of variable bounds, constraint right-hand sides, objective values, and constraint matrix coefficients. However, it could happen that by scaling constraints or variables, some constraint coefficients become too small. Note that Gurobi will treat any constraint coefficient with absolute value under 10^{-13} as zero. Consider the following example:

$$\begin{aligned} 10^{-7}x + 10y &\leq 10 \\ x + 10^4 z &\leq 10^3 \\ x, y, z &\geq 0, \end{aligned}$$

In this example, the matrix coefficients range in $[10^{-7}, 10^4]$. If we multiply all x coefficients by 10^5, and divide all coefficients in the second constraint by 10^3, we obtain:

$$\begin{aligned} 10^{-2}x' + y &\leq 10 \\ 10^2 x' + 10z &\leq 1 \\ x', y, z &\geq 0, \end{aligned}$$

where $x = 10^5 x'$. The resulting matrix coefficients have a range in $[10^{-2}, 10^2]$. Essentially the trick is to *simultaneously* scale a column and a row to achieve a smaller range in the coefficient matrix.

We recommend that you scale the matrix coefficients so that their range is contained in six orders of magnitude or less, and hopefully within $[10^{-3}, 10^6]$.

Avoid hiding large coefficients

As we said before, a typical recommendation for improving numerics is to limit the range of constraint matrix coefficients. The rationale behind this guideline is that terms to be added in a linear expression should be of comparable magnitudes so that rounding errors are minimized. For example:

$$\begin{aligned} x - 10^6 y &\geq 0 \\ y &\in [0, 10] \end{aligned}$$

is usually considered a potential source of numerical instabilities due to the wide range of the coefficients in the constraint. However, it is easy to implement a simple (but useless) alternative:

$$x - 10y_1 \geq 0$$

$$\begin{aligned}
y_1 - 10y_2 &= 0 \\
y_2 - 10y_3 &= 0 \\
y_3 - 10y_4 &= 0 \\
y_4 - 10y_5 &= 0 \\
y_5 - 10y &= 0 \\
y &\in [0, 10]
\end{aligned}$$

This form certainly has nicer values in the matrix. However, the solution $y = -10^{-6}$, $x = -1$ might still be considered feasible (within tolerances). A better alternative is to reformulate

$$\begin{aligned}
x - 10^6 y &\geq 0 \\
y &\in [0, 10]
\end{aligned}$$

as

$$\begin{aligned}
x - 10^3 y' &\geq 0 \\
y' &\in [0, 10^4]
\end{aligned}$$

where $10^{-3}y' = y$. In this setting, the most negative values for x which might be considered feasible would be -10^{-3}, and for y it would be -10^{-9}, which is a clear improvement over the original situation.

Dealing with big-M constraints

Big-M constraints are a regular source of instability for optimization problems. They are so named because they typically involve a large coefficient M that is chosen to be larger than any reasonable value that a continuous variable or expression may take. Here's a simple example:

$$\begin{aligned}
x &\leq 10^6 y \\
x &\geq 0 \\
y &\in \{0, 1\},
\end{aligned}$$

Big-M constraints are typically used to propagate the implications of a binary, on-off decision to a continuous variable. For example, a big-M might be used to enforce the condition that an edge can only admit flow if you pay the fixed charge associated with opening the edge, or a facility can only produce products if you build it. In our example, note that the $y = 0.0000099999$ satisfies the default integrality tolerance (`IntFeasTol`$=10^{-5}$), which allows x to take a value of 9.999. In other words, x can take a positive value without incurring an expensive fixed charge on y, which subverts the intent of only allowing a non-zero value for x when the binary variable y has the value of 1. You can reduce the effect of this behavior by adjusting the `IntFeasTol` parameter, but you can't avoid it entirely.

However, if the modeler has additional information that the x variable will never be larger than 10^3, then you could reformulate the earlier constraint as:

$$\begin{aligned}
x &\leq 10^3 y \\
x &\geq 0 \\
y &\in \{0, 1\}
\end{aligned}$$

And now, $y = 0.0000099999$ would only allow for $x \leq 0.01$.

For cases when it is not possible to either rescale variable x or tighten its bounds, an SOS constraints or an indicator constraint (of the form $y = 0 \Rightarrow x = 0$) may produce more accurate solutions, but often at the expense of additional processing time.

24.4 Does my model have numerical issues?

You can follow these steps to help determine whether a model is experiencing 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:

   ```
   Record 1
   ```

   Then, run your Gurobi program, which will produce `gurobi.rec` files. Afterwards, you can replay this recording file using `gurobi_cl`.

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

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

   The output will look like:

   ```
   Statistics for model (null) :
     Linear constraint matrix    : 25050 Constrs, 15820 Vars, 94874 NZs
     Variable types              : 14836 Continuous, 984 Integer
     Matrix coefficient range    : [ 0.00099, 6e+06 ]
     Objective coefficient range : [ 0.2, 65 ]
     Variable bound range        : [ 1, 5e+07 ]
     RHS coefficient range       : [ 1, 5e+07 ]
   ```

   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 10^9; smaller is better.

   In this example, the matrix range is

   $$6 \cdot 10^6 / 0.00099 = 6.0606 \cdot 10^9.$$

3. If possible, re-solve the model using the same 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: Model contains large matrix coefficient range
         Consider reformulating model or setting NumericFocus parameter
         to avoid 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. When the `optimize` function completes, print solution statistics. With the Python shell, use code like the following:

```
m.printQuality()
```

which provides a summary of solution quality:

```
Solution quality statistics for model Unnamed :
  Maximum violation:
    Bound       : 2.98023224e-08 (X234)
    Constraint  : 9.30786133e-04 (C5)
    Integrality : 0.00000000e+00
```

Violations that are larger than the tolerances are another indication of numerical issues. Also, for a pure LP (without integer variables), print the condition number via the following Python command:

```
m.KappaExact
```

The condition number measures the potential for error in linear calculations; a large condition number, such as 10^{12}, is another indication of possible numerical issues, see this section for more details.

5. If changing parameters (e.g., Method or Seed) leads to a different optimization status (e.g., `Infeasible` instead of `optimal`), or if the optimal objective values changes, this is usually a sign of numerical issues. To further assess this you can tighten tolerances (to the order of 10^{-8} or even 10^{-9}), and see if the behavior of the solver becomes consistent again. Note that tightening tolerances usually comes at the price of more computing time, and should not be considered as a solution for numerical issues.

24.5 Solver parameters to manage numerical issues

Reformulating a model may not always be possible, or it may not completely resolve numerical issues. When you must solve a model that has numerical issues, some Gurobi parameters can be helpful. We discuss these now, in descending order of relevance.

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 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 the resulting model is still numerically problematic, you may need to disable presolve completely using the parameter `Presolve=0`; try the steps above using

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

If the statistics look better with `Aggregate=0` or `Presolve=0`, you should further 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 relaxations of mixed-integer models: barrier and simplex.

The barrier algorithm is usually fastest for large, difficult models. However, it is also more numerically sensitive. And even when the barrier algorithm converges, the crossover algorithm that usually follows can stall due to numerical issues.

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

Note that, in many optimization applications, not all problem instances have numerical issues. Thus, choosing simplex exclusively may prevent you from taking advantage of the performance advantages of the barrier algorithm on numerically well-behaved instances. In such cases, you should use the concurrent optimizer, which uses multiple algorithms simultaneously and returns the solution from the first one to finish. The concurrent optimizer is the default for LP models, and can be selected for MIP by setting the Method parameter to 3 or 4.

For detailed 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 multiple distinct computers using distributed optimization. On a single computer, the different algorithms run on multiple threads, each using different processor cores. With distributed optimization, independent computers run the separate algorithms, which can be faster since the computers do not compete for access to memory.

Making the algorithm less sensitive

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

`ScaleFlag, ObjScale (All models)`: It is always best to reformulate a model yourself. However, for 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 use the largest objective coefficient to choose the scaling. 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. Settings 1-3 increasingly shift the focus towards more care in numerical computations, which can impact performance. The `NumericFocus` parameter employs a number of strategies to improve numerical behavior, including the use of quad precision and a tighter `Markowitz` tolerance. It is generally sufficient to try different values of `NumericFocus`. However, when `NumericFocus` helps numerics but makes everything much slower, you can 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 can contribute to numerical issues. Setting `GomoryPasses=0` may help numerics, but it may make the MIP more difficult to solve.

`Cuts (MIP)`: In some MIP models, various cuts can 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 addressing numerical issues. Numerical issues are better handled through model model reformulation.

24.6 Instability and the geometry of optimization problems

As we have seen, whenever we solve a problem numerically, we have to accept that the input we provide and the output we obtain may differ from the *theoretical* or *mathematical* solution to the given problem. For example, 0.1, in a computer, will be represented by a number that differs from 0.1 by about 10^{-17}. Thus, a natural thing to worry about is if these small differences may induce large differences in the computed solution.

This is the idea behind the notion of the *Condition Number* for a given problem. While it is true that for most practical optimization problems, small perturbations in the input only induce small perturbations in the final answer to the problem, there are some special situations where this is not the case. These ill behaving problems are called *Ill Conditioned* or *Numerically Unstable*.

This sections aims to show, in the context of linear optimization problems, the most common sources for this behavior, and also how to avoid the behavior altogether. We will review first the problem of solving linear systems with unique solutions, and then move into the more central issue of linear optimization problems, its geometric interpretation, and then describe some of the most common bad cases. We then provide two thought experiments with interactive material to help illustrate the concepts of this section. We conclude with some further thoughts on this topic.

Note that although the notion of the *Condition Number* has received a lot of attention from the academic community, reviewing this literature is beyond the scope of this document. If you want to start looking into this topic, a good entry point can be the Condition Number page at Wikipedia.

The case of linear systems:

Solving linear systems is a very common sub-routine in any MI(QC)P-solver, as we have to solve many linear systems during the full execution of the algorithm.

So, consider that we have a linear system $Ax = b$ with an unique solution (i.e. A is a square matrix with full rank), and you want to evaluate how the solution to the system might change if we perturb the right-hand side b. Since the system has a unique solution, we know that given b, the solution will be $A^{-1}b$, and if we perturb b with ε, the solution will be $A^{-1}(b+\varepsilon)$. A measure for the *relative* change in the solution with respect to the *relative* change in the input would be the ratio

$$\eta(b,\varepsilon) := \frac{\|A^{-1}b\|}{\|A^{-1}(b+\varepsilon)\|} \Big/ \frac{\|b\|}{\|b+\varepsilon\|}.$$

Note that the above definition is independent of the magnitudes of b and ε. From there, the *worst* possible ratio would be the result of

$$\kappa(A) := \max_{b,\varepsilon} \eta(b,\varepsilon).$$

This quantity is known as the condition number of the matrix A. It is not hard to prove that

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}},$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the maximum and minimum, respectively, eigenvalues of A. Equivalently

$$\kappa(A) = \frac{\|A\|}{\|A^{-1}\|}.$$

A common interpretation of $\kappa(A) = 10^k$ is that, when solving the system $Ax = b$, you may lose up to k digits of accuracy in x from the accuracy you have in b.

The condition number for the optimal simplex basis in an LP is captured in the `KappaExact` attribute. A very large κ value *might* be an indication that the result might be unstable.

When this is indeed the case, the best advice is to scale the constraint matrix coefficients so that the resulting range of coefficients is small. This transformation will typically reduce the κ value of the final basis; please refer to the Scaling section for a discussion on how to perform this rescaling, and also for caveats on scaling in general.

The geometry of linear optimization problems

Before showing optimization models that exhibit bad behavior, we first need to understand the *geometry* behind them. Consider a problem of the form

$$\begin{array}{ll} \max & cx \\ s.t. & Ax \leq b. \end{array}$$

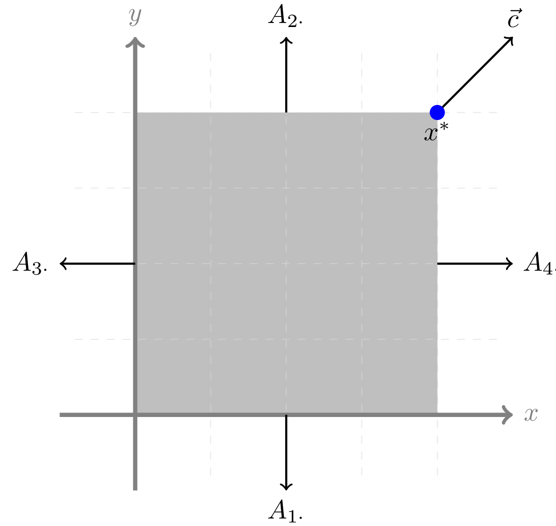
For example:

$$\begin{array}{lrrl} \max & x + y & \vec{c} = & (1,1) \\ s.t. & -x \leq 0 & A_{1\cdot} = & (-1,0) \\ & x \leq 1 & A_{2\cdot} = & (1,0) \\ & -y \leq 0 & A_{3\cdot} = & (0,-1) \\ & y \leq 1 & A_{4\cdot} = & (0,1). \end{array}$$

Note that if we denote $b^t := (0, 1, 0, 1)$, then the problem can be stated as

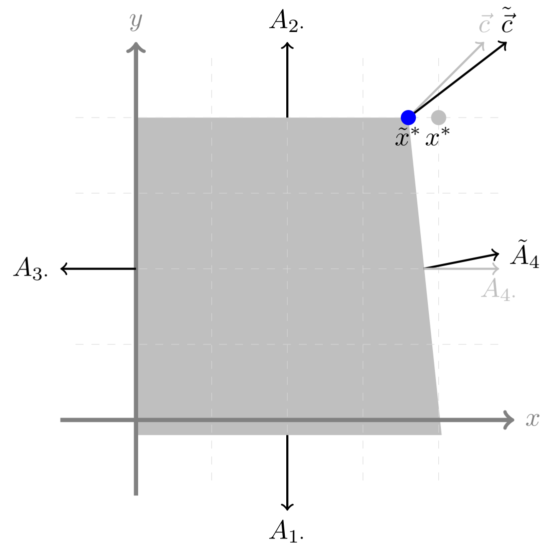
$$\max_{x \in \mathbb{R}^2} \{\vec{c}x : Ax \leq b\}.$$

The feasible region, direction of improvement $\vec{c}$, and optimal solution x^* can be depicted as



Note that whenever we move in the direction of $\vec{c}$, the value $\vec{c}x$ increases. Furthermore, since we can not move from x^* to another feasible point with better objective value, we can conclude that x^* is indeed the optimal solution for the problem. Note that x^* is a *corner* point of the feasible region. This is not a coincidence; you will always find an optimal solution at a corner point if the feasible region is bounded and $\vec{c}$ is not zero. If the objective is zero then all feasible solutions are optimal; we will talk more about zero objectives and their implications later.

To understand how changes in the input data affect the feasible region and the optimal solution, consider a small modification: $\tilde{b}^t = (\varepsilon, 1, 0, 1)$, $\tilde{\vec{c}} = (1+\varepsilon, 1)$, and $\tilde{A_{4\cdot}} = (\varepsilon, 1)$. Then our optimization problem would look like



Note that although we changed the right-hand side, this change had no effect in the optimal solution to the problem, but it did change the feasible region by enlarging the bottom part of the feasible area.

Changing the objective vector tilts the corresponding vector in the graphical representation. This of course also changes the optimal objective value. Perturbing a constraint tilts the graphical

representation of the constraint. The change in $A_{4\cdot}$ changes the primal solution itself. The amount of *tilting* constraint undergoes depends on the relative value of the perturbation. For example, although the constraint $x \leq 1$ and the constraint $100x \leq 100$ induce the same feasible region, the perturbation $x + \varepsilon y \leq 1$ will induce more tilting that the perturbation $100x + \varepsilon y \leq 100$.

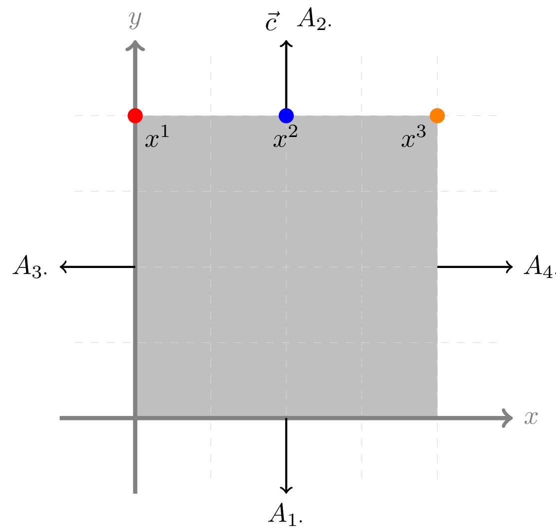
Multiple optimal solutions

A common misconception among beginners in optimization is the idea that optimization problems really have just one solution. Surprisingly, this is typically not true. For many practical problems, the objective (whether it is cost or revenue or ...) is dominated by a handful of variables, while most variables are just there to ensure that the actual *operation* of the solution is possible. Consider a staffing problem, for example, where cost is typically driven by the number of people who work on a given day, not by the specific people.

These kind of situations naturally lead to problems similar to

$$
\begin{array}{llrl}
\max & y & \vec{c} = & (0,1) \\
s.t. & -x \leq 0 & A_{1\cdot} = & (-1,0) \\
 & x \leq 1 & A_{2\cdot} = & (1,0) \\
 & -y \leq 0 & A_{3\cdot} = & (0,-1) \\
 & y \leq 1 & A_{4\cdot} = & (0,1).
\end{array}
$$

Graphically this can be depicted as



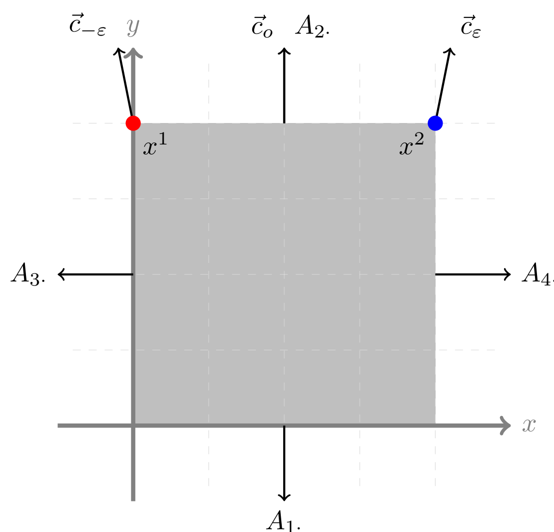
In this situation is clear that x^1, x^3, and all solutions lying on the line between these two points are optimal. The simplex algorithm will return either x^1 or x^3 (and may switch if you change parameters). The barrier algorithm (without crossover) will return x^2. These solutions are all correct; the problem as stated has no reason to prefer one over the other. If you do have a preference, you'll need to state it in your objective function.

Dealing with epsilon-optimal solutions

The previous section considered the case of multiple (true) optimal solutions. What happens when we have several ε-optimal solutions? To be more specific, consider

$$\begin{array}{llrl} \max & \varepsilon x + y & \vec{c} = & (\varepsilon, 1) \\ s.t. & -x \leq 0 & A_{1\cdot} = & (-1, 0) \\ & x \leq 1 & A_{2\cdot} = & (1, 0) \\ & -y \leq 0 & A_{3\cdot} = & (0, -1) \\ & y \leq 1 & A_{4\cdot} = & (0, 1). \end{array}$$

Graphically this can be depicted as



If ε is zero, then we are in the situation described before. Note, however, that a small perturbation of the objective vector may lead to either x^1 or x^2 being reported as optimal. And tolerances can play a big role here. If ε is negative, for example, then x^1 would be the mathematically optimal result, but due to the *optimality tolerance*, simplex might conclude that x^2 is optimal. More precisely, if ε is less than the default optimality tolerance of 10^{-6}, then simplex is free to declare either solution optimal (within tolerances).

The above statement is true whenever the *distance* between x^1 and x^2 is not too large. To see this, consider what happens when we change the right-hand side of $A_{4\cdot}$ from 1 to 10^6. Then the feasible region would then be a very long rectangular box, with vertices $(0, 0)$, $(0, 1)$, $(10^6, 1)$ and $(10^6, 0)$. Perhaps somewhat surprisingly, if ε is below the dual tolerance, simplex may consider $(10^6, 1)$ optimal, even though its objective value is $1 - 10^6\varepsilon$, which can be very relevant in terms of the final objective value.

Note that both situations share one ingredient: The objective function is (almost) parallel to one of the sides of the feasible region. In the first case, this side is relatively short, and thus jumping from x^2 to x^1 translate into a small change in objective value. In the second case, the side almost parallel to the objective function is very long, and now the jump from x^2 to x^1 can have a significant impact on the final objective function.

If you take out either of these two ingredients, namely the objective vector being almost parallel to a constraint, or the *edge* induced by this nearly-parallel constraint being very long, then this

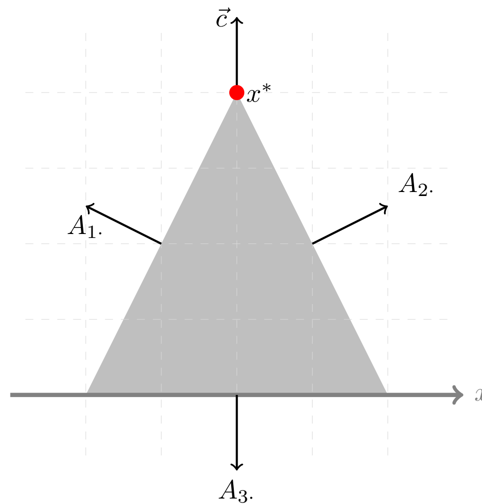
problem can not arise. For the reasons discussed at the beginning of this section, it is common for the objective function to be close to parallel to one or more constraints. Thus, the best way to avoid this situation is to avoid the second condition. The simplest way to do this is to ensure that the ranges for your variables are not too large. Please refer to the Scaling section for guidance on this.

Thin feasible regions

We now consider another extreme situation that can lead to unexpected results. Consider the problem defined as

$$\begin{array}{llll} \max & y & \vec{c} = & (0,1) \\ s.t. & -x+\varepsilon y \leq 1 & A_{1\cdot} = & (-1,\varepsilon) \\ & x+\varepsilon y \leq 1 & A_{2\cdot} = & (1,\varepsilon) \\ & -y \leq 0 & A_{3\cdot} = & (0,-1) \end{array}$$

and its graphical representation



It is clear from the graphical representation that the optimal solution for the problem will be at the intersection of constraints $A_{1\cdot}$ with $A_{2\cdot}$; and if we do the algebra, we will get that $x^* = (0, \frac{1}{\varepsilon})$. Also note that as you decrease ε the feasible region stretches upwards, leaving its base unchanged. We will consider the case where ε is a very small, positive number (between 10^{-9} and 10^{-6}).

If we perturb the right-hand side vector b from $(1,1)$ to $(1+\delta,1)$, the new solution will be $\tilde{x}^* = (-\frac{\delta}{2}, \frac{2+\delta}{2\varepsilon})$. To assess the impact of this perturbation, we compute the L_1 distance between this modified solution and the previous solution, which is given by

$$\|x^* - \tilde{x}^*\|_1 = \frac{|\delta|}{2} + \frac{|\delta|}{\varepsilon}$$

This quantity can be either small or very large, depending on the relative magnitude between δ and ε. If δ is much smaller than ε, then this quantity will be small. However, if δ is larger than or even the same order of magnitude as ε, the opposite will be true. Very small perturbations in the input data can lead to big changes in the optimal solution.

A similar issue arises if we perturb $A_{1\cdot}$ to $(-1, \delta)$; the new optimal solution becomes $\tilde{x}^* = (1 - \frac{2\varepsilon}{\varepsilon+\delta}, \frac{2}{\varepsilon+\delta})$. But now, if $\delta = \varepsilon/2$, then the new solution for y will change from $\frac{1}{\varepsilon}$ to $\frac{4}{3\varepsilon}$ (a 33% relative difference). Again, small changes in the input can produce big changes in the optimal solution.

What is driving this bad behavior? The problem is that the optimal point is defined by two constraints that are nearly parallel. The smaller ε is, the closer to parallel the are. When the constraints are so close parallel, small changes in the slopes can lead to big movements in the point where they intersect. Mathematically speaking:

$$\lim_{\varepsilon \to 0^+} \|x^*\| = \infty$$

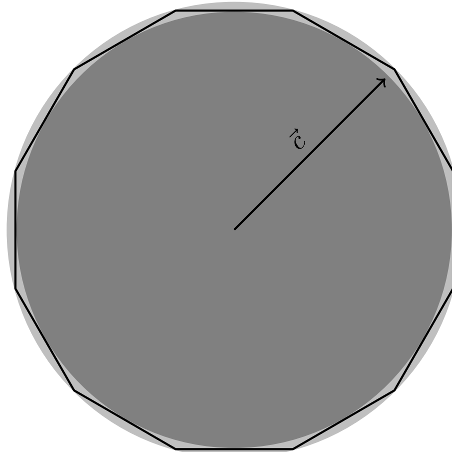
Note however that, if the original problem had an additional variable bound of the form $y \leq 10^4$, then neither of these bad behavior would have been possible. For any ε value smaller than 10^{-4}, the optimal point would be defined by the new constraint and one of the constraints $A_{2\cdot}$ or $A_{1\cdot}$, which would lead again to a well-behaved (i.e. stable) solutions. In summary, this sort of issue can only arise when either the feasible region is either unbounded or very large. See the Scaling section for further guidance on bounding the feasible region.

Optimizing over the circle:

Now we provide our first thought experiment: Consider the problem of optimizing a linear function over the feasible region defined by the constraints

$$\sin(2\pi \frac{i}{10^6})x + \cos(2\pi \frac{i}{10^6})y \leq 1, \forall i \in \{1, \ldots, 10^6\},$$

i.e. the feasible region is essentially a unit circle in $\mathbb{R}^2$. Note that for all objective functions, the corresponding optimal point will be defined by two linear constraints that are very close to be parallel. What will happen to the numerical solution to the problem? Can you guess? The situation is depicted in the figure below:



To perform the experiment, we execute the code `circleOpt.py`, where we randomly select an objective vector, find the optimal solution to the resulting optimization problem, and compute several relevant quantities:

- The worst *distance* between the reported primal solution, and the theoretical solution to the problem of actually optimizing over a perfect circle, over all previous runs.
- The worst bound violation reported by Gurobi over all previous runs.
- The worst constraint violation reported by Gurobi over all previous runs.
- The worst dual violation reported by Gurobi over all previous runs.
- The number of previous experiments.
- Accumulated number of simplex iterations.
- The κ (KappaExact atribute) value for the current optimal basis.

Sample output is shown below:

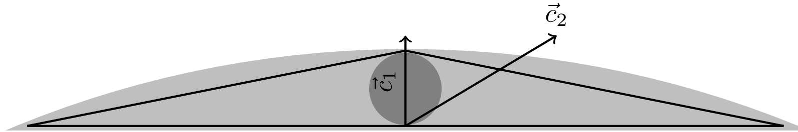
```
Added 2 Vars and 1048576 constraints in 19.19 seconds
Errors: 8.65535e-08 0 2.94137e-07 2.77556e-17 Iter 0 10 Kappa 3150.06
Errors: 4.81978e-07 0 3.22359e-07 2.77556e-17 Iter 1 21 Kappa 3009.12
Errors: 4.81978e-07 0 3.4936e-07 1.11022e-16 Iter 2 33 Kappa 2890.58
Errors: 1.53201e-06 0 9.78818e-07 1.11022e-16 Iter 6 79 Kappa 1727.89
Errors: 1.61065e-06 0 8.26005e-07 1.11022e-16 Iter 46 536 Kappa 1880.73
Errors: 1.61065e-06 0 8.84782e-07 1.11022e-16 Iter 52 602 Kappa 1817.27
Errors: 1.61065e-06 0 9.4557e-07 1.11022e-16 Iter 54 625 Kappa 1757.96
Errors: 1.69167e-06 0 9.78818e-07 1.11022e-16 Iter 64 742 Kappa 1727.89
Errors: 1.69167e-06 0 3.8268e-07 1.66533e-16 Iter 88 1022 Kappa 2761.99
Errors: 1.69167e-06 0 9.04817e-07 1.66533e-16 Iter 92 1067 Kappa 1797.06
Errors: 1.69167e-06 0 2.94137e-07 2.22045e-16 Iter 94 1089 Kappa 3150.06
Errors: 1.69167e-06 0 3.29612e-07 2.22045e-16 Iter 95 1101 Kappa 2975.84
Errors: 1.69167e-06 0 3.4936e-07 2.22045e-16 Iter 98 1137 Kappa 2890.58
Errors: 1.69167e-06 0 9.25086e-07 2.22045e-16 Iter 99 1147 Kappa 1777.3
Errors: 1.69167e-06 0 9.78818e-07 2.22045e-16 Iter 107 1237 Kappa 1727.89
Errors: 1.69167e-06 0 9.99895e-07 2.22045e-16 Iter 112 1293 Kappa 1709.61
Errors: 1.84851e-06 0 9.78818e-07 2.22045e-16 Iter 132 1523 Kappa 1727.89
Errors: 1.96603e-06 0 9.99895e-07 2.22045e-16 Iter 134 1545 Kappa 1709.61
```

Surprisingly the reported errors are rather small. Why is this? There are at least two contributing factors: the model has a bounded feasible region (in this case the range of both variables is $[-1, 1]$). In addition, the distance from one extreme point (a point at the intersection of two neighboring constraints) to its neighbor is also relatively small, so all ε-optimal solutions are close to each other.

We encourage you to play with this code, perturb some of the input data, and analyze the results. You will see the discrepancies between the theoretical and the numerical optimal solution will be comparable to the sizes of the perturbations.

Optimizing over thin regions:

Now we move to our second thought experiment: Consider a feasible region consisting of a triangle in $\mathbb{R}^2$ with a very wide base and very short height, as depicted here:



Consider the case where the ratio of the base to the height is on the order of 10^5, and that we consider a *nominal* objective function $\vec{c}_1$ as in the figure.

In theory, the optimal solution should be the apex of the triangle, but assume that we randomly perturb both the right-hand side and the objective function with terms in the order of 10^{-6}. What will happen with the numerical solution?

To perform the experiment, we execute the code `thinOpt.py`, where we perform a series of re-optimizations with different perturbations as described above. To be more precise, whenever the new computed solution is further from the mathetical solution by more than it has been in previous trials, we print:

- The new maximum distance among solutions.
- The current iteration.
- The κ (`KappaExact` atribute) value for the current optimal basis.
- The bound violation as reported by Gurobi for the current solution.
- The constraint violation as reported by Gurobi for the current solution.
- The dual violation as reported by Gurobi for the current solution.

Sample output is shown below:

```
New maxdiff 4e+16 Iter 0 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 1 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 2 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 7 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 83 Kappa 3.31072 Violations: 0 0 2.64698e-23
New maxdiff 4e+16 Iter 194 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 1073 Kappa 3.31072 Violations: 0 1.13687e-13 0
New maxdiff 4e+16 Iter 4981 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 19514 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 47117 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 429955 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 852480 Kappa 3.31072 Violations: 0 0 0
```

Results look very different from what we saw in our first test. The distance between the solution to the unperturbed model and the solution to the perturbed one is huge, even from the very first iteration. Also, the κ values are relatively small, and the reported primal, dual, and bound violations are almost zero. So, what happened? Note that when we choose $\vec{c}_1 = (0, 1)$, we are choosing an optimal point where a small tilting of the objective function may move us to another extreme point very far away, and hence the large norm. This is possible because the region is very large and, in principle, without any bounds, i.e. this is related to the case of ε-optimal solutions and very long sides.

Again, we encourage you to play with this example. For example, what would happen if the nominal objective function is $\vec{c}_2 = (1, 0)$?

Stability and convergence

The algorithms used to solve linear programming problems are all forced to make an assumption: that tiny changes to the system (e.g., making a small step in barrier) lead to small changes in the solution. If this is not true (due to ill-conditioning), then the algorithm may jump around in the solution space and have a hard time converging.

Finally, one way to improve the geometry of a problem is by suitably scaling variables and constraints as explained in the Scaling section, and working with bounded feasible sets with *reasonable* ranges for all variables.

24.7 Further reading

- *A Characterization of Stability in Linear Programming*, Stephen M. Robinson, 1977, Operations Research 25-3:435–447.
- *IEEE Standard for Binary Floating-Point Arithmetic* (IEEE 754), IEEE Computer Society, 1985.
- *What every computer scientist should know about floating-point arithmetic*, David Golberg, 1991, ACM Computing Surveys (CSUR), 23:5–48.
- *Numerical Computing with IEEE Floating Point Arithmetic*, Michael L. Overton, SIAM, 2001.
- *Practical guidelines for solving difficult linear programs*, Ed Klotz and Alexandra M. Newman, 2013, Surveys in Operations Research and Management Science, 18-1:1–17.
- *Identification, Assesment, and Correction of Ill-Conditioning and Numerical Instability in Linar and Integer Programs*, Ed Klotz, Bridging Data and Decisions, Chapter 3, 54–108.

Source code examples:

Source code for the experiment of optimizing over a circle

```
from gurobipy import *
from math import *
import random
import time
import sys

# Work on a circle defined on a million constraints
t0      = time.time()
n       = 1024 * 1024
m       = Model('Circle Optimization')
X       = m.addVars(2,lb=-2,ub=2)
Wb      = 0
Wc      = 0
Wd      = 0
maxdiff = 0
niter   = 0
margin  = 1.01
```

```
m.addConstrs(X[0]*cos((2*pi*i)/n) + X[1]*sin((2*pi*i)/n) <= 1
             for i in range(n))
print('Added 2 Vars and %d constraints in %.2f seconds' %
      (n, time.time()-t0))
m.Params.OutputFlag = 0
m.Params.Presolve   = 0

# Now select random objectives and optimize. Resulting optimal
# solution must be in the circle
for i in range(4096):
  theta=2*pi*random.random()
  a = cos(theta)
  b = sin(theta)
  m.setObjective(X[0] * a + X[1] * b)
  m.optimize()
  niter  += m.IterCount

  # See how far is the solution from the boundary of a circle of
  # radius one, if we minimize (a,b) the optimal solution should be (-a,-b)
  error  = (X[0].X+a)*(X[0].X+a) + (X[1].X+b)*(X[1].X+b)

  # Display most inacurate solution
  if (error > margin * maxdiff  or
      m.BoundVio > margin * Wb  or
      m.ConstrVio > margin * Wc or
      m.DualVio > margin * Wd     ):
    maxdiff = max(maxdiff, error)
    Wb      = max(Wb, m.BoundVio)
    Wc      = max(Wb, m.ConstrVio)
    Wd      = max(Wd, m.DualVio)
    print('Errors: %g %g %g %g Iter %d %d Kappa %g' %
          (maxdiff, Wb, Wc, Wd, i, niter, m.KappaExact))
    sys.stdout.flush()
```

Source code for the experiment on a thin feasible region

```
from gurobipy import *
import random
import sys

# Test the effect of small perturbations on the optimal solutions
# for a problem with a thin feasible region
rhs = 1e3
m   = Model('Thin line Optimization')
x   = m.addVar(obj=1)
y   = m.addVar(obj=0, lb=-GRB.INFINITY, ub=GRB.INFINITY)
c1  = m.addConstr(  1e-5 * y + 1e-0 * x <= rhs)
c2  = m.addConstr(- 1e-5 * y + 1e-0 * x <= rhs)
m.Params.OutputFlag = 0
m.Params.Presolve   = 0
m.optimize()
xval    = x.X
yval    = y.X
maxdiff = 0
for i in range(1024*1024):
  c1.Rhs = rhs + 2e-6 * random.random()
```

```
  c2.Rhs  = rhs + 2e-6 * random.random()
  x.Obj   = 1   + 2e-6 * random.random()
  y.Obj   = 0   + 2e-6 * random.random()
  m.optimize()
  x2val = x.X
  y2val = y.X
  error = (xval-x2val)*(xval-x2val) + (yval-y2val)*(yval-y2val)
  if error > 1e-5 + maxdiff:
    print('New maxdiff %g Iter %d Kappa %g Violations: %g %g %g' %
          (error, i, m.KappaExact, m.BoundVio, m.ConstrVio,
           m.DualVio))
    sys.stdout.flush()
    maxdiff = error
```

Source code for the experiment with column scalings

```
import sys
import random
import argparse
from gurobipy import *

# Use parameters for greater flexibility
parser = argparse.ArgumentParser(formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('-f','--infile', help='Problem File',
                    default=None, required=True)
parser.add_argument('-s','--scale', help='Scaling Factor',
                    type=float, default=10000.0)
parser.add_argument('-w','--outfile', help='Save scaled model',
                    default=None)
parser.add_argument('-o','--optimize', help='Optimize scaled problem',
                    type=int, default=1)
args = parser.parse_args()

# Load input problem
m = read(args.infile)

# Scale domain of all columns randomly in the given domain
for var in m.getVars():
  if var.vtype == GRB.CONTINUOUS:
    scale = random.uniform(args.scale/2.0, args.scale*2.0)
    flip = random.randint(0,3)
    if flip == 0:
      scale = 1.0
    elif flip == 1:
      scale = 1.0/scale
    col = m.getCol(var)
    for i in range(col.size()):
      coeff = col.getCoeff(i)
      row   = col.getConstr(i)
      m.chgCoeff(row, var, coeff*scale)
    var.obj = var.obj*scale
    if var.lb > -GRB.INFINITY:
      var.lb   = var.lb/scale
    if var.ub < GRB.INFINITY:
      var.ub   = var.ub/scale
```

```
if args.outfile != None:
  m.write(args.outfile)

# Optimize
if args.optimize:
  m.optimize()
  if m.Status == GRB.OPTIMAL:
    print('Kappa: %e\n' % m.KappaExact)
```