Gurobi Optimization LLC

Customer Applications

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

Gurobi Guidelines for Numerical Issues

...or why things go wrong

... and how to avoid it

... and how to avoid it

What are numerical issues?

Numerical issues is a generic name given to cases where the results of an optimization problem are either erratic, inconsistent, unexpected, or plain poor performance of the underlying algorithms.

- Rounding
- Real numbers aren't real
- Unrealistic expectations on precision
- Ill conditioning

Avoid Rounding Coefficients

And why it can be extremely dangerous

Consider the following input:
$$\begin{eqnarray*} x - 6y &=&1\\ 0.333x - 2y &= & .333 \end{eqnarray*}$$

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

And then, $x=1$ and $y=0$ is the only solution

Avoid Rounding Coefficients

If instead, we input:
$$ \begin{eqnarray*} x - 6y&=&1\\ 0.3333333333333333x - 2y&=&0.3333333333333333 \end{eqnarray*} $$

$$ \begin{eqnarray*} x - 6*(0.1666666666666667x - 0.1666666666666667) &=& 1\\ \Leftrightarrow 2\cdot10^{-16} x + 1 + 2\cdot10^{-16} &\approx& 1 \end{eqnarray*} $$

i.e. all solutions satisfying $x = 6y +1$ are accepted as feasible.

.... what about $2\cdot10^{-16}$?

Real numbers are not Real

Really really...

In [10]:

```
1 + 1e-16 == 1
```

Out[10]:

```
True
```

In [11]:

```
1e16 + 1 == 1e16
```

Out[11]:

```
True
```

The same is true for Excel, R, C, Java

The same is true for Excel, R, C, Java....

In [18]:

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

Out[18]:

```
False
```

... of course there are some exceptions... most notably **GNU-MP** and **GNU-bc**, but....

In [19]:

```
import sys
print(sys.float_info)
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.2250738585072014e-308, min_
exp=-1021, min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

What does this means?

- Most platforms provide dedicated hardware to handle IEEE 754 64 bit floating point numbers.
- Alternative **software representations** can be much slower than standard **double precision** numbers. (10x and 100x are not unusual)
- Gurobi (as most scientific computing software) uses **double precision** numbers.
- Testing for equality or exact computations simply won't do.

Tolerances and User-Scaling

Will always **solve given model**....

Presolve, or cutting planes, **may** convert

- $(1+10^{-7})x\leq 1,\,x\in\{0,1\}$

into

- $x=0$

Tolerances and User-Scaling

Will always **solve given model**.... but when a feasible solution is found

Checking for optimality always requires to check for:
Integrality:
We have to decide when a number x is an integer or not. Gurobi defines `IntInfTol` as the integrality tolerance, and accept x as integer if $\mathrm{abs}(x - \mathrm{floor}(x + 0.5))\leq\mathtt{IntInfTol}$

Primal Feasibility:
Given a linear constraint $a^t\cdot x\leq b$, we have to decide when that is satisfied or not. Gurobi defines `FeasibilityTol` as the primal feasibility tolerance, and accept x as primal feasible for a constraint if $a^t\cdot x -b \leq\mathtt{FeasibilityTol}$

Dual Feasibility:
Given a dual constraint $a^t \cdot y \leq c$, we have to decide when that is satisfied or not. Gurobi defines `OptimalityTol` as the dual feasibility tolerance, and accept y as a feasible **dual solutions** if $a^t\cdot y - c \leq\mathtt{OptimalityTol}$.

Tolerances and User-Scaling

What does this imply?

- Tolerances are **absolute** values.
  - Scaling affect the meaning of these.
- Given a point x^* feasible to the original model **within tolerances**
  - Presolve, cutting planes, **may** eliminate it from the feasible region
    - i.e. declare this particular solution `infeasible`
  - It may be found and by accepted during the solve process
- Thus ... for numerically unstable models
  - Even the optimal value may not be unique!

How bad are these limits?

- **Default** tolerances are 10^{-6} for primal/dual and 10^{-5} for integrality.
- If constraints range in 10^3 or 10^4, then primal feasible solutions will have a relative error of less than 10^{-9}, or one in a billion!
  - This is usually more than what is measurable in physical quantities
  - In engineering problems, just measurement/approximation error is **way** more than this.
  - Be aware of very small **active ranges** for constraints

- Same applies to range of variables.
  - Specially for integer variables... if range is around 10^6... do you really care if they are integer?

- For objective functions, **good solutions** should have objective values in the range $[-10^4:10^4]$.
  - Otherwise dual feasibility can become an issue
- Should also avoid optimal objective values too close to zero...
  - ... Unless objective is truly zero.

All this can **easily** be achieved by simply scaling rows, columns, and the objective function.

Getting to tight ranges for variables and constraints

There are three simple steps that help to get to good ranges:

- Use information not available to the solver to derive tight bounds
- Change units (eg. from tons to thousand tons) to get to good ranges
- Dissaggregate hierarchical objectives by using multi-objective optimization

Why Scaling is Relevant

Suppose we randomly scale columns in a given problem (within a range):

In [ ]:

```
# %load -r 31-40 rescale.py
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
```

Why Scaling is Relevant

In [6]:

```
%run rescale.py -f pilotnov.mps.bz2 -s 1
```

```
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Coefficient statistics:
  Matrix range     [2e-06, 1e+07]
  Objective range  [2e-03, 1e+00]
  Bounds range     [5e-06, 6e+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: 0.03s
Presolved: 721 rows, 1659 columns, 11454 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -3.2008682e+05   1.451454e+05   0.000000e+00      0s
     991   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 991 iterations and 0.14 seconds
Optimal objective -4.497276188e+03
Kappa: 2.681961e+06
```

Why Scaling is Relevant

In [7]:

```
%run rescale.py -f pilotnov.mps.bz2 -s 1000
```

```
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Coefficient statistics:
  Matrix range     [5e-09, 9e+09]
  Objective range  [2e-06, 2e+03]
  Bounds range     [5e-09, 8e+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: 0.02s
Presolved: 875 rows, 1917 columns, 11899 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -6.1727412e+32   7.113559e+31   6.172741e+02      0s
    1395   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1395 iterations and 0.21 seconds
Optimal objective -4.497276188e+03
Kappa: 2.740228e+07
```

Why Scaling is Relevant

In [8]:

```
%run rescale.py -f pilotnov.mps.bz2 -s 1e6
```

```
Optimize a model with 975 rows, 2172 columns and 13057 nonzeros
Coefficient statistics:
```

```
  Matrix range     [6e-12, 9e+12]
  Objective range  [2e-08, 1e+06]
  Bounds range     [5e-12, 3e+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 99 rows and 250 columns
Presolve time: 0.02s
Presolved: 876 rows, 1922 columns, 11920 nonzeros

Iteration    Objective       Primal Inf.    Dual Inf.      Time
       0   -7.9535394e+34   7.293786e+31   7.953539e+04      0s
    1555   -4.4972762e+03   0.000000e+00   0.000000e+00      0s

Solved in 1555 iterations and 0.22 seconds
Optimal objective -4.497276188e+03
Warning: unscaled primal violation = 2320.84 and residual = 0.119918
Kappa: 3.230371e+11
```

Why Scaling is Relevant

In [2]:

```
%run rescale.py -f pilotnov.mps.bz2 -s 1e8
```

```
Optimize a model with 975 rows, 2172 columns and 13056 nonzeros
Coefficient statistics:
  Matrix range     [1e-13, 8e+14]
  Objective range  [3e-10, 1e+08]
  Bounds range     [5e-14, 5e+12]
  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 86 rows and 249 columns
Presolve time: 0.02s

Solved in 0 iterations and 0.02 seconds
Infeasible model
```

Advanced User-Scaling

Row or objective scaling can be considered simultaneously with **column** scaling:

Consider
$$ \begin{eqnarray*} 10^{-7}x + y &\leq& 0\\ x - 10^{3} z & \leq & 10^4\\ x&\in&[0,10^5] \end{eqnarray*} $$

This can be replaced with $$\begin{eqnarray*} 10^{-3}x' + y &\leq& 0\\ 10x' + z &\leq&10\\ x'&\in&[0,10], \end{eqnarray*}$$ where $x=10^4x'$.

Note that solving $10x'+z\leq10$ is easier than $10^4x'+10^3z\leq10^4$!

Don't lie to yourself

Usual guideline: Keep range of coefficients of rows/columns small:

$$\begin{eqnarray*} x - 10^{6} y &\geq& 0 \\ y&\geq&0 \end{eqnarray*} $$

$$\begin{eqnarray*} x - 10 y_1 &\geq& 0\\ y_1 - 10 y_2 &=& 0\\ y_2 - 10 y_3 &=& 0\\ y_3 - 10 y_4 &=& 0\\ y_4 - 10 y_5 &=& 0\\ y_5 - 10 y &=& 0\\ y&\geq&0 \end{eqnarray*}$$

Still has $y=-10^{-6}$ and $x=-1$ as feasible!

Don't lie to yourself

Usual guideline: Keep range of coefficients of rows/columns small:

$$\begin{eqnarray*} x - 10^{6} y &\geq& 0 \\ y&\geq&0 \end{eqnarray*} $$

If $y\in[0,10]$ we can use: $$ \begin{eqnarray*} x - 10^{3} y' &\geq& 0 \\ y'&\in&[0,1e^4]\\ \end{eqnarray*} $$ where $10^{-3} y' = y$.

In this setting the most negative values for x would be -10^{-3}, and for y it would be -10^{-9}.

Big-M constraints

Constructs such as: $$ \begin{eqnarray*} x&\leq&10^6y\\ x&\geq&0\\ y&\in& \{0,1\} \end{eqnarray*} $$

Widely used... **BUT**... $x = 9.9999, y = 0.0000099999$ is feasible

$$ \begin{eqnarray*} x&\leq&10^3y\\ x &\geq& 0\\ y &\in & \{0,1\} \end{eqnarray*} $$
And now, $y = 0.0000099999$ would only allow for $x \leq 0.01$.

Otherwise... try **SOS** or **General Implication** constraints.

In Summary

- Try to get range of variables within reason ($[-10^4:10^4]$)
- Try to select scaling of rows so ***active range*** is within reason ($[-10^4:10^4]$)
- Try to scale objective so that ***good*** solutions are in the range ($[-10^4:10^4]$)
- Don't expect to compare quantities with very different absolute values (above 10 orders of magnitude...)

- This may improve performance of your current model!
  - Gurobi does try to clean up numerical issues
  - The cost... running time
  - Extra steps
  - Disable some routines

Does my model have numerical issues?

- Isolate the issue:
  - Use `GURO_PAR_DUMP 1`
  - Will generate `gurobi.rew` and `gurobi.prm`

- Inspect the model:

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

```
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 ]
```

```
Matrix coefficient range     : [ 0.00099, 5e+00 ]
Objective coefficient range : [ 0.2, 65 ]
Variable bound range        : [ 1, 5e+07 ]
RHS coefficient range       : [ 1, 5e+07 ]
```

- What is a reasonable range?

Does my model have numerical issues?

- Review the logs:

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

- Usual warning outputs:

```
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 = ...
```

Does my model have numerical issues?

- Inspect solution quality:

```
m.printquality()
```

- Summary of violations:

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

- Inspect the Condition Number:

```
m.KappaExact
```

Does my model have numerical issues?

- Other symptoms:
  - If I change some parameter the solution changes!
  - If I change the seed the solution changes!
  - If I change algorithm the model becomes infeasible!
  - If I change tolerances...

- *99.9%* is a sign of numerical problems:
  - If by tightening tolerances, things become stable...
  - If you already scaled the ranges for primal and dual values...
    - Give us call!

Solver parameters to *manage* numerical issues

Presolve

- Sometimes presolve may deteriorate numerics
  - Try:

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

- If increase range:
  - Use `Presolve=0`
  - Use `Aggregate=0`
  - Use `AggFill=0`

Use the right algorithm:

- *Usually* Barrier is fastest, but more sensitive to numerics
  - Try different root methods:
    - Simplex based: `Method=0 Method=1`
    - Barrier: `Method=2`
    - Concurrent: `Method=3 Method=4 Method=5`

Make the algorithm less sensitive:

- `NumericFocus`: Modifies a series of parameters that improve numeric behavior, this includes:
  - `MarkowitzTol` for simplex
  - `Quad` for extended precision simplex
  - `GomoryPasses`
  - `CutPasses`
  - `CutAggPasses`
- `NormAdjust`: Manages pricing in Simplex.
- `BarHomogeneous`: For infeasible/unbounded models, this may help Barrier.
- `CrossoverBasis`: Usually slower, but more robust crossover for Barrier.
- `Cuts`: Disable cuts may help numerical stability.

Advanced Material

What is the Condition Number?

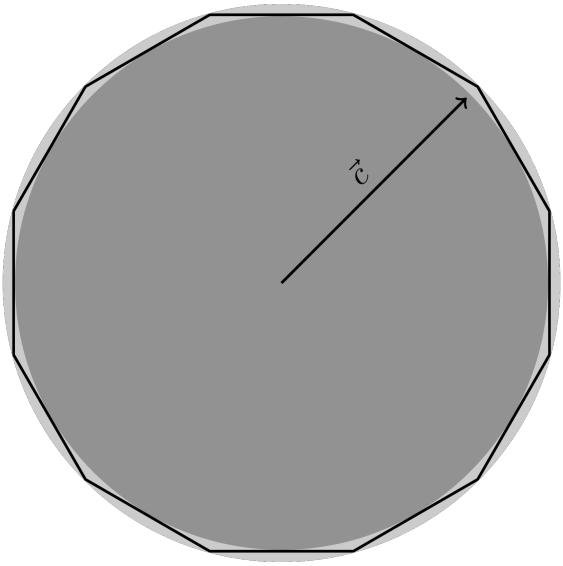
Not all issues with instability of results have to do with numeric errors. One broad concept is the **sensibility** of a given problem to perturbations in the input data. This concept, usually known as **Condition Number**, is relevant because in the presence of numeric

perturbations in the input data. This concept, usually known as **Condition Number**, is relevant because in the presence of numeric errors, this indicates how likely it is to get unstable answers.

For Linear systems $Ax=b$, The condition number is denoted and defined as $$\kappa(A):=\|A\|\cdot\|A^{-1}\|$$

A common **geometric** interpretation of this concept has to do with how **rounded** the feasible region is. For bounded sets, you can think of it as the ratio between the smallest **circle** containing the feasible region and the largest circle contained in the feasible region.

Optimizing over a circle



Optimizing over a circle

In [2]:

```
from gurobipy import *
from math import *
n = 1024
m = Model('Circle Optimization')
X = m.addVars(2,lb=-2,ub=2)
m.addConstrs(X[0]*cos((2*pi*i)/n) + X[1]*sin((2*pi*i)/n) <= 1 for i in range(n))
m.update()
```

```
Warning for adding constraints: zero or small (< 1e-13) coefficients, ignored
```

Optimizing over a circle

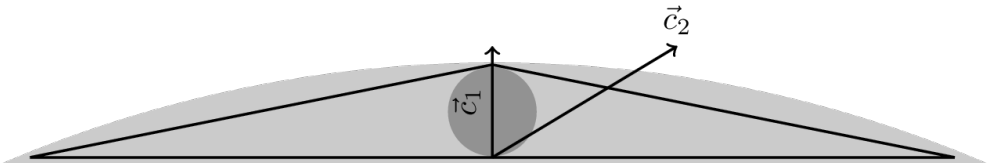
In [3]:

```
%run circleOpt.py
```

```
Warning for adding constraints: zero or small (< 1e-13) coefficients, ignored
Added 2 Vars and 1048576 constraints in 20.62 seconds
Errors: 2.99567e-09 0 4.74059e-07 1.38778e-17 Iter 0 10 Kappa 2481.79
Errors: 1.30387e-07 0 9.66287e-07 5.55112e-17 Iter 1 21 Kappa 1739.05
Errors: 5.82762e-07 0 3.12808e-07 6.93889e-17 Iter 2 33 Kappa 3054.68
Errors: 5.82762e-07 0 2.94137e-07 1.11022e-16 Iter 3 44 Kappa 3150.06
Errors: 5.82762e-07 0 9.78818e-07 1.66533e-16 Iter 5 67 Kappa 1727.89
Errors: 6.92325e-07 0 3.4936e-07 1.66533e-16 Iter 11 133 Kappa 2890.58
Errors: 6.92325e-07 0 3.8268e-07 1.66533e-16 Iter 15 179 Kappa 2761.99
Errors: 1.08973e-06 0 9.99895e-07 1.66533e-16 Iter 16 190 Kappa 1709.61
Errors: 1.11311e-06 0 9.4557e-07 1.66533e-16 Iter 225 2562 Kappa 1757.96
Errors: 1.11311e-06 0 9.99895e-07 1.66533e-16 Iter 286 3257 Kappa 1709.61
Errors: 1.31624e-06 0 9.29161e-07 1.66533e-16 Iter 291 3313 Kappa 1773.4
Errors: 1.31624e-06 0 9.99895e-07 1.66533e-16 Iter 304 3464 Kappa 1709.61
```

```
Errors: 1.43708e-06 0 8.84782e-07 1.66533e-16 Iter 320 3648 Kappa 1817.27
Errors: 1.43708e-06 0 9.00796e-07 1.66533e-16 Iter 330 3764 Kappa 1801.07
Errors: 1.62804e-06 0 9.66287e-07 1.66533e-16 Iter 333 3796 Kappa 1739.05
Errors: 1.62804e-06 0 9.99895e-07 1.66533e-16 Iter 335 3817 Kappa 1709.61
Errors: 1.62804e-06 0 2.94137e-07 2.22045e-16 Iter 359 4079 Kappa 3150.06
Errors: 1.62804e-06 0 3.27188e-07 2.22045e-16 Iter 361 4102 Kappa 2986.85
Errors: 1.62804e-06 0 3.34495e-07 2.22045e-16 Iter 362 4113 Kappa 2954.05
Errors: 1.62804e-06 0 8.84782e-07 2.22045e-16 Iter 368 4181 Kappa 1817.27
Errors: 1.62804e-06 0 9.99895e-07 2.22045e-16 Iter 369 4192 Kappa 1709.61
Errors: 1.70968e-06 0 9.66287e-07 2.22045e-16 Iter 474 5387 Kappa 1739.05
Errors: 1.70968e-06 0 9.99895e-07 2.22045e-16 Iter 478 5431 Kappa 1709.61
```

Optimizing over a thin line



Optimizing over a thin line

In [6]:

```
from gurobipy import *
# 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.update()
```

Optimizing over a thin line

In [7]:

```
%run thinOpt.py
```

```
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 6.61744e-24
New maxdiff 4e+16 Iter 50 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 91 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 695 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 1692 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 1876 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 4462 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 4522 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 51461 Kappa 3.31072 Violations: 0 0 2.11758e-22
New maxdiff 4e+16 Iter 189604 Kappa 3.31072 Violations: 0 0 1.05879e-22
New maxdiff 4e+16 Iter 428283 Kappa 3.31072 Violations: 0 0 0
New maxdiff 4e+16 Iter 899680 Kappa 3.31072 Violations: 0 0 0
```

Condition Number

Recap

- Some instability is induced by the geometry of the problem **defined by the user**.
  - This instability may be exacerbated by limited precision.

- Since Gurobi does not second guess user input, these issues can become critical.
- Most of these geometric issues can (and should) be avoided via **reformulation**.
  - This reformulation must be done at the user-level

Thanks for your attention!

... Questions?

Further reading:

- IEEE Standard for Binary Floating-Point Arithmetic, also known as IEE 754
- 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.